Марк Г. Собель

Linux

Администрирование и системное программирование

Полное руководство по использованию команд, оболочек, утилит и редакторов в Linux и Mac OS

PRENTICE HALL

2-Е ИЗДАНИЕ

ПИТЕР®

A Practical Guide to Linux Commands, Editors, and Shell Programming

SECOND EDITION

Mark G. Sobell

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Linux

Администрирование и системное программирование

2-Е ИЗДАНИЕ

Марк Г. Собель

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2011

ББК 32.973.2-018.2
УДК 004.451
С54

Собель М.
С54 Linux. Администрирование и системное программирование. 2-е изд. — СПб.: Питер, 2011. — 880 с.: ил.

ISBN 978-5-459-00450-2

Эта книга представляет собой всеобъемлющее руководство по работе, настройке, администрированию и системному программированию операционных систем семейства UNIX/Linux, включая Ubuntu, Fedora, openSUSE, Red Hat, Debian, Mandriva, Mint и даже Mac OS X. Первая часть книги дает начальные представления о работе в Linux: объясняется интерфейс командной строки, приводится обзор наиболее распространенных утилит, рассматривается иерархическая файловая система Linux, объясняется, как использовать свойства оболочки, чтобы ускорить работу в системе. Во второй части рассмотрены два мощных Linux-редактора, работающих в режиме командной строки и включенных в большинство дистрибутивов Linux: текстовый редактор vim и GNU-редактор emacs. В третьей части книги освещаются аспекты работы с оболочками bash и tcsh. Четвертая часть посвящена важным средствам программирования, широко используемым при системном администрировании Linux и Mac OS: рассказывается об особенностях расширенного программирования оболочки с использованием bash; дается представление о создании сценариев администрирования Linux-систем на языке программирования Perl; подробно рассмотрены утилиты sed и rsync. Заключительная часть книги представляет собой обширный справочник команд Linux, включающий подробные примеры использования более сотни наиболее важных утилит.

Книга адресована широкому кругу читателей: от пользователей Linux и Mac OS X, желающих узнать преимущества работы с системой из командной строки, до системных администраторов и программистов, нуждающихся в более глубоком понимании системы и доступных им инструментальных средств.

ББК 32.973.2-018.2
УДК 004.451

Права на издание получены по соглашению с Prentice Hall, Inc. Upper Sadle River, New Jersey 07458. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0131367364 англ.
ISBN 978-5-459-00450-2

© Prentice Hall, Inc. 2010
© Перевод на русский язык ООО Издательство «Питер», 2011
© Издание на русском языке, оформление ООО Издательство «Питер», 2011

Оглавление

Предисловие **11**
Особенности данной книги 13
Содержание 14
Часть I. Операционные системы Linux и Mac OS X 14
Часть II. Редакторы 16
Часть III. Оболочки 16
Часть IV. Средства программирования 17
Часть V. Справочник команд 18
Вспомогательные материалы 19

Глава 1. Добро пожаловать в Linux и Mac OS X **20**
История UNIX и GNU–Linux 21
Что можно отнести к достоинствам Linux? 25
Обзор Linux 31
Дополнительные свойства Linux 37
Заключение 39
Упражнения 40

ЧАСТЬ I. ОПЕРАЦИОННЫЕ СИСТЕМЫ LINUX И MAC OS X

Глава 2. Начнем! **42**
Соглашения, используемые в этой книге 42
Вход в систему через терминал или эмулятор терминала 45
Работа с оболочкой 46
su/sudo: Ограничение ваших полномочий (root-права) 50
Где найти документацию 52
Дополнительная информация о входе в систему 60
Заключение 63
Упражнения 64

Глава 3. Утилиты **65**
Специальные символы 65
Основные утилиты 66
Работа с файлами 69
| (канал): Организация связи между процессами 76
Еще четыре утилиты 77
Сжатие и архивирование файлов 80
Команды для поиска мест, где находятся утилиты 85
Получение информации о пользователе и системе 87
Связь с другими пользователями 91
Электронная почта 92
Заключение 93
Упражнения 95

Глава 4. Файловая система ... 97
Иерархическая файловая система ... 97
Полные имена ... 103
Работа с каталогами ... 106
Права доступа ... 114
Списки управления доступом (ACL-списки) ... 121
Ссылки ... 126
Заключение ... 132
Упражнения ... 134

Глава 5. Оболочка ... 137
Командная строка ... 137
Стандартный ввод и стандартный вывод ... 143
Запуск команды в фоновом режиме ... 154
Генерация имени файла и расширение имени ... 157
Встроенные команды ... 161
Заключение ... 162
Упражнения ... 164

ЧАСТЬ II. РЕДАКТОРЫ

Глава 6. Редактор vim ... 168
История ... 168
Руководство. Использование vim для создания и редактирования файла ... 169
Введение в функции редактора vim ... 177
Командный режим: перемещение курсора ... 184
Режим ввода ... 188
Командный режим: Удаление и изменение текста ... 189
Поиск и замена ... 193
Другие команды ... 199
Копирование, перемещение и удаление текста ... 200
Чтение и запись файлов ... 202
Установка параметров ... 203
Расширенные технологии редактирования ... 208
Единицы отсчета ... 211
Заключение ... 214
Упражнения ... 218

Глава 7. Редактор emacs ... 220
История ... 220
Руководство по началу работы с emacs ... 223
Графический интерфейс пользователя emacs ... 230
Основные команды редактирования ... 231
Интерактивная справка ... 239
Расширенное редактирование ... 241
Основные режимы: редактирование с восприимчивостью к языкам программирования ... 256
Дополнительная информация ... 270
Заключение ... 271
Упражнения ... 278

ЧАСТЬ III. ОБОЛОЧКИ

Глава 8. Оболочка Bourne Again Shell 282

Предыстория 283
Основы оболочки 284
Параметры и переменные 304
Специальные символы 320
Процессы 321
История 324
Псевдонимы 341
Функции 344
Управление bash: Свойства и настройки 347
Обработка командной строки 351
Заключение 362
Упражнения 364

Глава 9. Оболочка TC Shell 367

Сценарии оболочки 368
Вход в оболочку TC Shell и выход из нее 369
Свойства, общие для оболочек Bourne Again Shell и TC Shell 371
Перенаправление стандартной ошибки 377
Работа с командной строкой 378
Переменные 384
Управляющие структуры 397
Встроенные команды 406
Заключение 411
Упражнения 412
Упражнения повышенной сложности 413

ЧАСТЬ IV. СРЕДСТВА ПРОГРАММИРОВАНИЯ

Глава 10. Программирование оболочки Bourne Again Shell 416

Управляющие структуры 417
Описатели файлов 449
Параметры и переменные 452
Встроенные команды 465
Выражения 480
Программы оболочки 488
Заключение 499
Упражнения 501
Упражнения повышенной сложности 502

Глава 11. Язык сценариев Perl 504

Введение в Perl 504
Переменные 512
Управляющие структуры 520
Работа с файлами 530
Сортировка 534
Подпрограммы 535
Регулярные выражения 538

Модули CPAN ... 544
Примеры ... 546
Заключение ... 550
Упражнения ... 550
Упражнения повышенной сложности ... 551

Глава 12. Язык обработки шаблонов AWK ... 552
Синтаксис ... 552
Аргументы ... 553
Ключи ... 553
Примечания ... 554
Основы языка ... 554
Примеры ... 562
Заключение ... 583
Упражнения ... 584
Упражнения повышенной сложности ... 584

Глава 13. Редактор sed ... 585
Синтаксис ... 585
Аргументы ... 585
Ключи ... 585
Основы редактора ... 586
Примеры ... 590
Заключение ... 600
Упражнения ... 601

Глава 14. Утилита безопасного копирования rsync ... 602
Синтаксис ... 602
Аргументы ... 602
Ключи ... 603
Примеры ... 605
Заключение ... 613
Упражнения ... 613

ЧАСТЬ V. СПРАВОЧНИК КОМАНД

Указатель команд ... 616
Общие ключи ... 619
sample ... 620
aspell ... 622
at ... 626
bzip2 ... 630
cal ... 632
cat ... 633
cd ... 635
chgrp ... 636
chmod ... 640
chown ... 645
cmp ... 647

comm ... 649
configure ... 651
cp ... 653
cpio ... 657
crontab ... 662
cut ... 665
date ... 667
dd ... 669
df ... 672
diff ... 674
diskutil (OS X) ... 680
ditto (OS X) ... 683
dmesg ... 685
dscl (OS X) ... 686
du ... 689
echo ... 691
expr ... 694
file ... 697
find ... 699
finger ... 706
fmt ... 708
fsck ... 710
ftp ... 715
gawk ... 722
gcc ... 723
GetFileInfo (OS X) ... 727
grep ... 729
gzip ... 734
head ... 737
kill ... 738
killall ... 740
launchctl (OS X) ... 741
less ... 744
ln ... 748
lpr ... 750
ls ... 753
make ... 761
man ... 767
mkdir ... 771
mkfs ... 772
Mtools ... 775
mv ... 778
nice ... 780
nohup ... 782
od ... 782
open (OS X) ... 786
otool (OS X) ... 788
paste ... 790
pax ... 792
plutil (OS X) ... 798
pr ... 799

ps 801
rcp 805
renice 807
rlogin 809
rm 810
rmdir 811
rsh 812
rsync 814
scp 814
sed 816
SetFile (OS X) 817
sleep 818
sort 820
split 829
ssh 830
stat 838
strings 839
stty 840
sysctl (OS X) 845
tail 846
tee 849
telnet 850
test 852
top 855
touch 859
tr 861
tty 864
tune2fs 864
umask 867
uniq 868
w 870
wc 872
which 873
who 875
xargs 877

Предисловие

Linux

В книге «Linux. Администрирование и системное программирование» объясняется, как работать с командной строкой операционной системы Linux. Первые несколько глав быстро введут в курс дела читателей, имеющих сравнительно небольшой опыт работы на компьютере. Остальная часть книги адресована опытным пользователям. В этой книге не дается описание конкретного выпуска или дистрибутива Linux, она подходит для всех последних версий этой операционной системы.

Mac OS X

Вы узнаете как работать с основой операционной системы Mac OS X, которая досталась ей от UNIX/Linux. Книга позволит вам «заглянуть под капот», обходя стороной традиционный графический интерфейс пользователя (GUI), который отождествляется большинством людей с компьютерами Macintosh, и объяснит, как использовать мощный интерфейс командной строки (command-line interface, CLI). В этой книге не приводится описание конкретного выпуска OS X, поскольку она подходит для всех последних выпусков. Материалы, относящиеся к Linux, также относятся и к Mac OS X. Различия между этими операционными системами оговариваются особо.

Интерфейс командной строки

Первым появился интерфейс командной строки (CLI), или текстовый интерфейс, позволяющий управлять операционной системой Linux прямо из командной строки. Тогда еще не было мыши, с помощью которой можно было указывать на элементы экрана или перетаскивать значки. В некоторых программах (например, emacs) были реализованы примитивные окна, использующие весьма скромные возможности псевдографического оформления, предоставляемые набором ASCII-символов. Выделение областей экрана осуществлялось с помощью негативного изображения.

Система Linux зарождалась и совершенствовалась именно в таком окружении, поэтому вполне естественно, что все исходные инструменты Linux вызывались из командной строки. В этой среде до сих пор скрывается реальная мощь Linux, чем и объясняется тот факт, что многие профессионалы Linux работают *исключительно* из командной строки. Благодаря четким описаниям и множеству примеров эта книга показывает, как добиться наибольшей отдачи от вашей Linux-системы, используя интерфейс командной строки.

Дистрибутив Linux

Дистрибутив Linux содержит ядро, утилиты и прикладные программы. В настоящее время доступно множество дистрибутивов — Ubuntu, Fedora, Red Hat, Mint, OpenSUSE, Mandriva, CentOS и Debian. Хотя дистрибутивы несколько отличаются друг от друга, все они основаны на ядре Linux, утилитах и приложениях. В этой книге взят за основу тот код, который является общим для большинства дистрибутивов, поэтому вы можете использовать его независимо от того, какой именно дистрибутив работает на вашей машине.

Новое в этом издании

Это издание содержит множество новых и обновленных материалов:

- В нем дан обзор интерфейса командной строки Mac OS X (по всей книге). В части V рассматриваются утилиты и подчеркивается разница между теми ключами утилит, которые используются при работе под управлением Linux, и теми ключами, которые используются при работе под управлением Mac OS X.
- Добавлена глава о языке сценариев Perl (глава 11).
- Обновлено описание утилиты безопасного копирования rsync (глава 14).
- В части V рассмотрено более пятнадцати новых утилит, включая некоторые утилиты, доступные только при работе под управлением Mac OS X.

Похожие источники информации

Если вам приходилось читать книгу «A Practical Guide to Red Hat® Linux®: Fedora™ and Red Hat Enterprise Linux» или «A Practical Guide to Ubuntu Linux®», вы можете заметить схожесть изложенных в них материалов с некоторыми материалами данной книги. В этих книгах очень похожи предисловия и главы, посвященные утилитам (речь идет о главе 3 данной книги, но не о части V), файловой системе, оболочке Bourne Again Shell (bash) и языку Perl. Вместе с тем, в данной книге есть главы, которых нет в вышеупомянутых книгах, — это главы, посвященные редакторам vim и emacs, оболочке TC Shell (tcsh), языкам AWK и sed, утилите rsync, а также часть V, в которой подробно описываются 97 наиболее востребованных утилит Linux и полезных программ Mac OS X.

Круг читателей

Эта книга адресована широкому кругу читателей. Для ее чтения не нужен богатый опыт программирования, но предполагается наличие некоторого опыта в использовании компьютерных систем. Она вполне подойдет:

- **студентам**, изучающим предметы, в которых предусмотрено использование Linux или Mac OS X;
- **опытным пользователям**, желающим исследовать все преимущества работы с Linux или Mac OS X из командной строки;
- **профессионалам**, использующим Linux или Mac OS X в своей работе;
- **начинающим пользователям Macintosh**, желающим узнать, что такое UNIX/Linux и почему все продолжают говорить об их важности, а также как пользоваться их преимуществами;
- **опытным пользователям Macintosh**, желающим узнать, как воспользоваться всей мощью UNIX/Linux, лежащей в основе Mac OS X;
- **пользователям UNIX**, желающим применить свое мастерство работы с UNIX к среде Linux или Mac OS X;
- **системным администраторам**, нуждающимся в более глубоком понимании Linux или Mac OS X и доступных им инструментальных средств, включая оболочку bash и язык сценариев Perl;
- **студентам, изучающим компьютерные дисциплины**, осваивающим операционную систему Linux или Mac OS X;
- **программистам**, которым нужно разобраться в программной среде Linux или Mac OS X;
- **техническим специалистам**, желающим постичь основы Linux или Mac OS X.

Преимущества данной книги

«Linux. Администрирование и системное программирование» позволяет всецело усвоить вопросы использования Linux или Mac OS X из командной строки. Независимо от уровня ваших исходных знаний, данная книга предлагает сведения, необходимые в вашей практической деятельности. Прочитав эту книгу, вы поймете, как использовать Linux/OS X, а содержимое книги послужит ценным справочником на многие последующие годы.

Для систем Macintosh всегда было доступно огромное количество свободного программного обеспечения. К тому же большую активность проявляет сообщество пользователей условно-бесплатных программ для Macintosh. Представляя UNIX/Linux аспекты, присущие Mac OS X, эта книга открывает для пользователей Macintosh огромные запасы свободного и дешевого программного обеспечения, применяемого в Linux и других UNIX-подобных систем.

В ДАННОЙ КНИГЕ ТАМ, ГДЕ ГОВОРИТСЯ О LINUX, ИМЕЕТСЯ В ВИДУ НЕ ТОЛЬКО LINUX, НО И MAC OS X

Операционная система UNIX является общим предком как для Linux, так и для Mac OS X. Несмотря на существенные различия в графических пользовательских интерфейсах (GUI) этих двух операционных систем, их интерфейсы командной строки (CLI) очень похожи друг на друга и во многих случаях совпадают. В этой книге описываются интерфейсы командных строк как Linux, так и Mac OS X. Чтобы упростить чтение книги, термин Linux используется в ней применительно и к Linux, и к Mac OS X. Там, где между этими двумя операционными системами имеются различия, даются специальные пояснения.

Особенности данной книги

Структура книги рассчитана на легкое извлечение нужной информации в различных ситуациях. Например, ее можно прочитать от корки до корки, изучив с нуля все, что касается использования командной строки Linux. Но при наличии достаточного опыта в работе с Linux можно использовать эту книгу как справочное пособие: достаточно будет найти интересующую тему в оглавлении или указателе и прочитать обо всем, что ее касается. Или же перейти к описанию одной из утилит в части V, «Справочник команд».

Можно также рассматривать эту книгу в качестве каталога Linux. В книге также имеется множество указателей на веб-сайты, из которых можно извлечь дополнительную информацию: считайте, что Интернет является своеобразным расширением данной книги.

Книга «Linux. Администрирование и системное программирование» обладает следующими особенностями:

- Дополнительные разделы, позволяющие читать книгу на разных уровнях, возвращаясь к более сложному материалу после предварительной подготовки к его усвоению.
- Врезки ВНИМАНИЕ, обращающие ваше внимание на те процедуры, при проведении которых можно легко ошибиться, и подсказывающие правильные действия, предупреждая возникновение проблем.
- Врезки СОВЕТ, в которых даются сведения, позволяющие сэкономить время за счет применения более рациональных подходов, или приводится дополнительная информация, представляющая какую-нибудь пользу или интерес.

- Врезки БЕЗОПАСНОСТЬ, указывающие способы повышения безопасности системы.
- Веб-сайт www.sobell.com, поддерживающий книгу и включающий различные исправления, загружаемые примеры, приводимые в книге, указатели на полезные веб-сайты и ответы на упражнения с четными номерами.
- Практические примеры, которые приводятся во всех частях книги и иллюстрируют изложенный в книге материал.
- Множество полезных URL (интернет-адресов), ведущих на сайты, где можно получить программное обеспечение и информацию.
- Заключения для каждой из глав, выделяющие наиболее важные из рассмотренных вопросов.
- Завершающие каждую главу упражнения для читателей, желающих отточить свое мастерство. Ответы на упражнения с четными номерами доступны на веб-сайте www.sobell.com.
- Подробные описания важных инструментальных средств, созданных в рамках проекта GNUs, включая gcc, систему конфигурации и сборки GNU, make, gzip, и множество других инструментов.
- Указатели, помогающие извлечь онлайн-документацию из множества источников, включая локальную систему и Интернет.
- Подробные описания важных утилит командной строки, разработанных компанией Apple специально для операционной системы Mac OS X, включая diskutil, ditto, dscl, GetFileInfo, launchctl, otool, plutil и SetFile.

Содержание

В этом разделе описывается информация, излагаемая в каждой главе, и объясняется, как эта информация способствует извлечению имеющихся в Linux преимуществ. Для получения более подробных сведений можно просмотреть оглавление.

- Глава 1, «Добро пожаловать в Linux и Mac OS X».

  Здесь представлена предыстория появления на свет Linux и OS X. В главе дается историческая справка по Linux, краткая справка по ядру OS X Mach, объясняется, как проект GNU помог Linux на первых порах, и рассматривается ряд важных свойств Linux, отличающих эту операционную систему от других.

Часть I. Операционные системы Linux и Mac OS X

ВОЗМОЖНО, ОПЫТНЫМ ПОЛЬЗОВАТЕЛЯМ БУДЕТ ДОСТАТОЧНО ВСЕГО ЛИШЬ БЕГЛО ПРОСМОТРЕТЬ ЧАСТЬ I

Если с системой UNIX/Linux уже приходилось иметь дело, возможно, будет вполне достаточно просмотреть или пропустить некоторые или все главы первой части данной книги. Раздел «Соглашения, используемые в этой книге» на с. 42, в котором объясняются принятые в книге соглашения по типографскому оформлению текста, а также раздел «Где найти документацию» на с. 52, в котором указываются как локальные, так и удаленные источники документации по Linux, нужно просмотреть всем читателям.

В Части I дается представление об операционной системе Linux и начальные сведения о работе с ней.

- Глава 2, «Начнем!».

  Здесь объясняются соглашения по типографскому оформлению книги, помогающие представит материал в более понятном виде и упростить чтение книги. Эта глава предоставляет базовую информацию и объясняет, как войти в систему, изменить свой пароль, дать системе Linux команду с помощью оболочки и найти документацию по системе.

- Глава 3, «Утилиты».

  Здесь объясняется интерфейс командной строки и дается краткое введение в более чем 30 утилит, использующих данный интерфейс. Изучение этой главы позволяет понять, что такое операционная система Linux, и узнать, что собой представляют некоторые инструментальные средства повседневного пользования. В этой главе рассматриваются следующие утилиты:
  - grep, осуществляющая поиск в файлах строки символов;
  - unix2dos, осуществляющая преобразование текстовых файлов Linux в формат Windows;
  - tar, создающая архивные файлы, которые могут содержать многие другие файлы;
  - bzip2 и gzip, сжимающие файлы, позволяя им занимать меньше места на диске, и позволяющие вам пересылать их по сети с более высокой скоростью;
  - diff, выводящая различия между двумя текстовыми файлами.

- Глава 4, «Файловая система».

  Здесь рассматривается иерархическая файловая система Linux, включая файлы, имена файлов, путевые имена, работу с каталогами, права доступа, а также жесткие и символьные ссылки. Знание файловой системы позволяет организовать свои данные таким образом, чтобы не тратить лишнее время на поиск нужной информации. Это дает также возможность делиться некоторыми своими файлами с другими пользователями, оставляя остальные файлы закрытыми от посторонних глаз.

- Глава 5, «Оболочка».

  Здесь объясняется, как использовать свойства оболочки, чтобы ускорить и облегчить свою работу. Все свойства, рассмотренные в этой главе, присущи как bash, так и tcsh. В этой главе рассматриваются следующие вопросы:
  - использование ключей командной строки для изменения способов работы команды;
  - внесение незначительных изменений в командную строку для перенаправления ввода в команду, чтобы он исходил из файла, а не с клавиатуры;
  - перенаправление вывода команды в файл вместо экрана;
  - использование каналов для отправки вывода из одной утилиты непосредственно другой утилите, чтобы все задачи решались с использованием только одной командной строки;
  - запуск программ в фоновом режиме, чтобы можно было работать с одной задачей, в то время как Linux будет работать с другой;
  - использование оболочки для генерации файловых имен с целью экономии времени на их набор и с целью оказания помощи в том случае, если пользователь забыл конкретное имя файла.

Часть II. Редакторы

Во второй части книги рассмотрены два мощных классических Linux-редактора, работающих в режиме командной строки. Текстовый редактор vim, «улучшенная» версия широко распространенного редактора vi, а также популярный GNU-редактор emacs включены в большинство дистрибутивов Linux. Текстовые редакторы позволяют создавать и изменять текстовые файлы, в которых могут содержаться программы, сценарии оболочки, памятки и данные для ввода в программы форматирования текста. Поскольку администрирование системы Linux предполагает редактирование конфигурационных файлов, имеющих текстовую основу, высокопрофессиональные Linux-администраторы имеют большой опыт по части использования текстовых редакторов.

- Глава 6, «Редактор vim».

  Глава начинается с руководства по редактору vim, далее в ней объясняется, как используются многие расширенные функции редактора, включая специальные символы в строках поиска, универсальный и именованные буферы, параметры, маркеры и выполнение команд непосредственно из самого редактора. Глава завершается кратким справочником по командам этого редактора.

- Глава 7, «Редактор emacs».

  Глава открывается руководством по работе с редактором, а затем в ней объясняются многие свойства редактора emacs, а также показывается, как используются клавиши META, ALT и ESCAPE. Дополнительно рассматривается привязка клавиш, буферы, а также полный поиск и поиск с последовательным уточнением, как с использованием символьных строк, так и с использованием регулярных выражений. Дается подробное описание взаимоотношений между Указателем, Курсором, Меткой и Областью. Также объясняется, как извлечь пользу из всеобъемлющих интерактивных справочных средств, доступных из редактора emacs.

  Другие рассматриваемые темы включают вырезку и вставку, использование нескольких окон и областей, работу с режимами emacs — особенно с режимом языка C, который помогает программистам создавать и отлаживать C-код. Глава 7 завершается кратким справочником по командам редактора emacs.

Часть III. Оболочки

Часть III посвящена более подробному рассмотрению оболочки bash и введению в оболочку TC Shell (tcsh).

- Глава 8, «Оболочка Bourne Again Shell».

  Здесь повествование продолжается с того места, на котором оно остановилось в главе 5, при этом рассматриваются более тонкие аспекты работы с оболочкой. Для примеров в этой главе используется Bourne Again Shell — bash, оболочка, практически всегда используемая для системных сценариев оболочки. В главе 8 описывается:

  - как использовать файлы загрузки, ключи и свойства оболочки для ее настройки;
  - как использовать управление заданиями для их приостановки и перемещения из активного в фоновый режим и обратно;

- как модифицировать и повторно выполнять команды, используя исторический список оболочки;
- как создавать псевдонимы для настройки команд;
- как в сценариях оболочки работать с переменными, созданными пользователем, и с переменными, являющимися ключевыми словами;
- как настраивать функции, аналогичные сценариям оболочки, но выполняемые значительно быстрее;
- как создавать и выполнять простые сценарии оболочки;
- как перенаправлять сообщения об ошибках, чтобы они попадали в файл, а не на экран.

- Глава 9, «Оболочка TC Shell».

  Здесь описывается оболочка tcsh, и рассматриваются как общие для bash и tcsh свойства, так и те свойства, которые отличают одну оболочку от другой. В этой главе объясняется:

  - как запустить tcsh и поменять оболочку, используемую по умолчанию на tcsh;
  - как перенаправить сообщения об ошибках, чтобы они попадали в файл, а не на экран;
  - как использовать управляющие структуры для изменения порядка выполнения сценариев оболочки;
  - как работать с числовыми переменными и переменными массивов tcsh;
  - как использовать встроенные команды оболочки.

Часть IV. Средства программирования

Часть IV посвящена важным средствам программирования, широко используемым при системном администрировании Linux и Mac OS, а также при создании программ общего назначения.

- Глава 10, «Программирование оболочки Bourne Again Shell».

  Здесь повествование продолжается с того места, на котором оно остановилось в главе 8, рассматриваются особенности расширенного программирования оболочки с использованием bash, подкрепляемые подробными примерами. В этой главе рассматриваются:

  - управляющие структуры, такие как if...then...else и case;
  - переменные, включая вопросы их локальности;
  - арифметические и логические (булевы) выражения;
  - некоторые, наиболее полезные встроенные команды оболочки, включая exec, trap и getopts.

  После усвоения основ Linux можно будет воспользоваться своим знаниями для создания более сложных и специализированных программ, используя в качестве языка программирования оболочку.

  В главе 10 даются две сложные задачи, связанные с программированием оболочки, а затем шаг за шагом показывается методика их решения. В первой задаче

используется рекурсия для создания иерархии каталогов. Во второй задаче разрабатывается программа quiz, показывающая, как создается сценарий оболочки, взаимодействующий с пользователем, и объясняется порядок обработки данных в сценарии. (Примеры в части V также показывают многие свойства утилит, которые можно использовать в сценариях оболочки.)

- Глава 11, «Язык сценариев Perl».

  Здесь дается представление о популярном языке программирования Perl, обладающим весьма широкими возможностями. В этой главе рассматриваются:

  - справочные средства Perl, включая perldoc;
  - переменные Perl и управляющие структуры;
  - обработка файлов;
  - регулярные выражения;
  - установка и использование CPAN-модулей.

  На языке Perl написаны многие сценарии администрирования Linux-систем. После чтения главы 11 вы сможете лучше разбираться в этих сценариях, а также приступить к созданию своих собственных сценариев. Эта глава включает множество примеров сценариев на языке Perl.

- Глава 12, «Язык обработки шаблонов AWK».

  Здесь объясняется, как создавать программы, использующие довольно мощный язык AWK, фильтрующий данные, записывающий отчеты и извлекающий данные из Интернета. В разделе расширенного программирования описывается, как установить двунаправленную связь с другой программой, используя сопроцессы, и как получить ввод по сети, а не из локального файла.

- Глава 13, «Редактор sed».

  Здесь дается описание утилиты sed, неинтерактивного потокового редактора, который находит множество применений в качестве фильтра внутри сценариев оболочки. В этой главе рассматривается порядок использования буферов sed для написания достаточно простых, но мощных программ и дается множество примеров.

- Глава 14, «Утилита безопасного копирования rsync».

  Здесь рассматривается rsync, надежная утилита, копирующая обычный файл или иерархию каталога на локальной машине или между локальной системой и другой системой, подключенной к сети. По мере написания программ, эту утилиту можно использовать для их резервного копирования на другие системы.

Часть V. Справочник команд

Система Linux включает сотни утилит. Главы 12–14, а также часть V предоставляют подробные примеры использования сотни наиболее важных утилит, с помощью которых можно решить проблемы, не обращаясь к программированию на языке C. Для тех, кто уже знаком с системами UNIX/Linux, эта часть книги послужит весьма объемным и простым в использовании справочником. Для недостаточно опытных пользователей она послужит полезным подспорьем при усвоении предыдущих частей книги.

Хотя описания утилит, предоставленные в главах 12–14, а также в части V, даны в формате, похожем на тот, который используется в справочных страницах Linux-руководства (man-страницах), они намного проще читаются и усваиваются. Описания именно этих утилит включены в книгу потому, что они широко востребованы в повседневной работе (например, ls и cp), являются мощными средствами, особенно полезными в сценариях оболочки (sort, paste и test), помогают работать с системой Linux (ps, kill и fsck) или позволяют связываться с другими системами (ssh, scp и ftp). Описание каждой утилиты включает исчерпывающее объяснение действий ее наиболее полезных ключей с указанием различий между ключами, поддерживаемыми при работе под управлением Mac OS X, и ключами, поддерживаемыми при работе под управлением Linux. Подразделы «Комментарии» и «Примечания» полны советов и приемов, позволяющих извлечь из применения утилит все, на что они способны. В подразделах «Примеры» показывается использование этих утилит в реальной жизни, как в одиночку, так и совместно с другими утилитами с целью составления отчетов, суммирования данных и извлечения информации. Чтобы увидеть, насколько объемны эти примеры, обратите внимание на разделы «Примеры» для языка AWK (более 20 страниц в главе 12), для утилиты ftp (в ее описании в части V) и для утилиты sort (также в части V).

Вспомогательные материалы

На личной странице автора (www.sobell.com) содержатся загружаемые листинги всех длинных программ, приведенных в данной книге, а также указатели на множество интересных и полезных сайтов, связанных с Linux и OS X, имеющихся во Всемирной паутине, перечень исправлений, ответы на все упражнения с четными номерами, а также дается приглашение на внесение уточнений, написание комментариев и высказывание предложений.

Я беру на себя всю ответственность за любые ошибки и упущения, допущенные в данной книге. Если вы их обнаружите или просто захотите что-то прокомментировать, поставьте меня в известность (mgs@sobell.com), и я внесу исправления в следующее издание. На моей личной странице (www.sobell.com) имеется перечень ошибок и упоминания о тех, кто их обнаружил. Там также имеются копии длинных сценариев из данной книги и указатели на множество интересных веб-страниц, посвященных Linux.

Марк Собель (Mark G. Sobell)
Сан-Франциско, Калифорния

Глава 1. Добро пожаловать в Linux и Mac OS X

Операционная система — это низкоуровневое программное обеспечение, занимающееся планировкой задач, распределением памяти и управлением интерфейсом с внешним оборудованием, например с принтерами, дисковыми приводами, экраном, клавиатурой и мышью. Операционная система состоит из двух основных частей: *ядра* и *системных программ*. Ядро занимается распределением ресурсов машины, включая память, дисковое пространство и циклы центрального процессора, между всеми другими программами, запущенными на компьютере. К системным программам относятся драйверы устройств, библиотеки, служебные программы (утилиты), оболочки (интерпретаторы команд), сценарии и файлы конфигурации, прикладные программы, серверы и программы документации. Они занимаются выполнением высокоуровневых вспомогательных задач, часто работая в качестве серверов в клиент-серверных структурах. Многие библиотеки, серверы и служебные программы были написаны в рамках проекта GNU, краткие сведения о котором будут даны ниже.

Ядро Linux

Ядро Linux было разработано финским студентом Линусом Торвальдсом (Linus Torvalds), который воспользовался Интернетом, чтобы тут же запустить исходный код во всеобщий свободный доступ. Торвальдс выпустил Linux версии 0.01 в сентябре 1991 года.

Новая операционная система стала результатом большого и напряженного труда. Программисты всего мира довольно быстро расширили ядро и разработали другие инструментальные средства, добавив наряду с новыми функциональными возможностями и те, что уже встречались в операционных системах BSD UNIX и System V UNIX (SVR4). Название Linux является комбинацией, составленной из слов Linus и UNIX.

Операционная система Linux, разработанная благодаря сотрудничеству многих людей по всему миру, является продуктом Интернета и относится к бесплатным операционным системам (имеющим открытый код). Другими словами, весь исходный код находится в свободном распространении. Его можно совершенно свободно изучать, распространять и изменять. В результате код доступен без каких-либо ограничений, то есть совершенно бесплатно, это касается программного обеспечения, исходного кода, документации и поддержки (через тематические конференции, подписки и другие интернет-ресурсы). Определение свободного программного обеспечения GNU (www.gnu.org/philosophy/free-sw.html) трактуется следующим образом:

Бесплатное пиво

В понятие «свободное программное обеспечение» вкладывается смысл свободы, а не цены. Чтобы вникнуть в это понятие, нужно думать о «свободе» как о «свободе слова», а не как о «свободном (то есть бесплатном) пиве».

Ядро Mach

Операционная система OS X запускает ядро Mach, разработанное в университете Карнеги-Меллона (Carnegie Mellon University, CMU), и является свободным программным обеспечением. CMU оформила свою работу в проект в 1994 году, хотя другие группы продолжали эту линию исследований. Much

относится к программному обеспечению Mac OS X и имеет открытый исходный код: ядро Mac OS X основано на коде Mach и FreeBSD, утилиты взяты из BSD и проекта GNU, а системные программы большей частью взяты из кода BSD, хотя компания Apple разработала и ряд новых программ.

LINUX, OS X И UNIX

Linux и OS X тесно связаны с операционной системой UNIX. В данной книге рассматривается Linux и OS X. Чтобы упростить чтение, в ней рассматривается Linux, а подразумевается OS X и Linux, а все случаи, когда OS X работает не так как Linux, оговариваются особо. По этой же причине термин Linux в данной главе часто используется для описания свойств обеих операционных систем, и Linux, и OS X.

История UNIX и GNU–Linux

Этот раздел посвящен предыстории, связывающей UNIX и Linux, а также отношениям между GNU и Linux. По адресу www.levenez.com/unix можно найти довольно любопытную историческую диаграмму, связанную с UNIX.

UNIX как источник наследственных черт Linux

Система UNIX была создана теми исследователями, которые нуждались в наборе современных компьютерных инструментов, помогающих в разработке их проектов. Эта система позволяла группе людей, ведущих совместную работу над проектом, использовать общие избранные данные и программы, сохраняя при этом конфиденциальность всей остальной информации.

Главную роль в дальнейшем росте популярности операционной системы UNIX сыграли университеты и колледжи благодаря так называемому «четырехлетнему эффекту». Когда в 1975 году к операционной системе UNIX был открыт широкий доступ, компания Bell Labs предложила ее образовательным учреждениям по весьма скромной цене. В свою очередь, школы использовали эту операционную систему в своих программах по изучению информатики, обеспечивая знакомство с ней всех, кто изучал данный предмет. Поскольку UNIX представляла собой передовую систему разработки, студенты стали осваивать довольно сложную среду программирования. После прихода на производство бывшие студенты предполагали, что будут работать в такой же передовой среде. По мере продвижения многих из них по карьерной лестнице в коммерческом мире, операционная система UNIX пробилась и в промышленную сферу.

BSD (Berkeley) UNIX

Кроме ознакомления студентов с операционной системой UNIX, исследовательская группа по компьютерным системам (Computer Systems Research Group, CSRG) Калифорнийского университета в Беркли разработала для нее существенные расширения и изменения. Фактически количество внесенных популярных изменений было столь велико, что одна из версий была названа дистрибутивом программного обеспечения системы UNIX, разработанным в Беркли — Berkeley Software Distribution (BSD) UNIX system (или просто Berkeley UNIX). Другой основной версией стала UNIX System V (SVR4), производная от версий, разработанных и поддерживаемых компанией AT&T и лабораторией UNIX

System Laboratories. Операционная система Mac OS X в своей основе унаследовала очень многие черты именно от ветви BSD этого дерева.

Эволюционное развитие до 1983 года

Ричард Столлман (Richard Stallman, www.stallman.org) анонсировал[1] Проект GNU, запущенный в целях разработки операционной системы, включавшей в себя как ядро, так и системные программы, и представил манифест GNU Manifesto[2], начинающийся следующими словами:

GNU, что означает «Gnu — это не UNIX» (Gnu's Not UNIX), является названием полностью UNIX-совместимой программной системы, которую я создал, поэтому я могу бесплатно делиться ею со всеми, кто может ее использовать.

Несколькими годами спустя, когда Столлман понял, что своим высказыванием он всех запутал, он добавил к предыдущему предложению следующее примечание:

Здесь дана несколько расплывчатая формулировка. Суть высказывания состояла в том, что никто не должен платить за разрешение использовать систему GNU. Но слова не прояснили ситуацию, и люди часто понимали это так, что копии GNU должны всегда распространяться с минимальными изменениями или вообще в неизменном виде. Я не вкладывал в высказывание именно такой смысл, чуть позже в манифесте упоминалась возможность для компаний предоставлять поддержку пакета распространяемых программ с целью получения прибыли. Впоследствии я понял, что нужно четко разграничить значение слова «free» в смысле свободы, от значения этого же слова в смысле цены. Программное обеспечение, предваряемое словом «free», предоставляет пользователям свободу по его распространению и изменению. Некоторым пользователям копии могут доставаться бесплатно, а некоторые будут платить за приобретение копий, и если денежные средства помогают поддержать процесс внесения усовершенствований в программное обеспечение, я не вижу в этом ничего плохого. Важно понять, что все обладатели копий могут свободно сотрудничать со всеми, кто их использует.

В манифесте после краткого описания сути проекта и текущих достижений Столлман приводит следующее продолжение:

Почему я должен создавать программы в рамках проекта GNU.

Я считаю, что золотое правило требует поделиться программой, которая мне нравится, со всеми, кому она тоже нравится. Продавцы программ хотят разобщить пользователей и добиться от них покорности, заставляя каждого согласиться с тем, что он не вправе делиться программами ни с кем другим. Я отказываюсь разрушать солидарность с другими пользователями подобным образом. Я не могу с чистой совестью подписывать договор о нераспространении или лицензионное соглашение. В течение многих лет я работал в лаборатории искусственного интеллекта, сопротивляясь подобным тенденциям и другим недружественным шагам, но со временем они зашли слишком далеко: я не мог оставаться в учреждении, где все это затрагивало меня помимо моей воли.

---

[1] www.gnu.org/gnu/initial-announcement.html

[2] www.gnu.org/gnu/manifesto.html

Поэтому, чтобы я мог и дальше бесплатно пользоваться компьютером, я решил объединить достаточное количество свободного программного обеспечения, чтобы получить возможность обойтись без любых платных компьютерных программ. Я ушел из лаборатории искусственного интеллекта, чтобы лишить MIT (Массачусетский технологический институт) каких-либо законных оснований чинить мне препятствия по свободному распространению GNU.

Сцена следующая, 1991 год

Проект GNU успешно продвигался к своей цели. Была завершена основная часть операционной системы GNU, за исключением ядра. Позже Ричард Столлман напишет следующее:

К началу 90-х мы скомпоновали всю систему кроме ядра (и мы также работали над ядром GNU Hurd[1], которое запускалось в виде надстройки над Mach[2]). Вопреки ожиданиям, разработка этого ядра давалась намного труднее, и мы все еще бились над ее завершением[3].

...Многие верили в то, что как только Линус Торвальдс завершит работу над ядром, его друзья подыщут другое свободное программное обеспечение, и при наличии всего самого необходимого ничто уже не сможет помешать сделать UNIX-подобную систему доступной.

И не случайно они нашли именно то, что нужно, — систему GNU. Доступное и свободное программное обеспечение[4] стало звеном, завершающим систему, поскольку проект GNU разрабатывался с этой целью с 1984 года. Цель разработки UNIX-подобной системы под названием GNU была сформулирована в самом манифесте GNU. В исходном анонсе проекта GNU также выделялся ряд оригинальных планов по разработке системы GNU. Ко времени написания Linux система [GNU] была уже практически завершена[5].

В наши дни «операционная система» GNU запускается в виде надстройки над ядрами FreeBSD (www.freebsd.org) и NetBSD (www.netbsd.org), соблюдая при этом полную совместимость с двоичными кодами Linux, а также в виде надстройки предварительных версий Hurd и Darwin (developer.apple.com/opensource), уже без поддержки совместимости.

Свободно распространяемый код

Традиция свободного кода берет свое начало с тех времен, когда UNIX продавалась университетам по символической цене, благодаря чему она приобрела свой успех и свойство переносимости. Со временем, в результате коммерциализации UNIX и обращения исходного кода в собственность производителей, что сделало его абсолютно недоступным, эта традиция тихо умерла. Еще одна проблема, касающаяся коммерче-

[1] www.gnu.org/software/hurd/hurd.html

[2] www.gnu.org/software/hurd/gnumach.html

[3] www.gnu.org/software/hurd/hurd-and-linux.html

[4] www.gnu.org/philosophy/free-sw.html

[5] www.gnu.org/gnu/linux-and-gnu.html

ских версий UNIX, была связана с их сложностью. Поскольку каждый производитель настроил UNIX под специфическую архитектуру, операционная система стала менее переносимой, а также слишком громоздкой для обучения и проведения различных экспериментов.

MINIX

Преследуя учебные цели, два профессора создали свои собственные упрощенные UNIX-подобные системы: Дуг Камер (Doug Comer) создал XINU, а Эндрю Таненбаум (Andrew Tanenbaum) создал MINIX. Линус Торвальдс создал Linux, чтобы избавиться от недостатков MINIX. Когда нужно было выбирать между простотой кода и его эффективностью и предоставляемыми возможностями, Таненбаум предпочитал простоту (чтобы облегчить преподавание с использованием MINIX), то есть у этой системы отсутствовали многие, необходимые людям свойства. Система Linux разрабатывалась исходя из совершенно противоположных взглядов.

Систему Linux можно совершенно бесплатно взять в Интернете. Можно также получить код GNU, воспользовавшись услугами почтовой службы США и заплатив незначительную сумму за материалы и доставку. Вы можете поддержать фонд свободного программного обеспечения — Free Software Foundation (www.fsf.org), купив этот же код (GNU), пересылаемый в бандеролях, имеющих более высокую цену, а также вы можете приобрести коммерческие выпуски Linux (называемые *дистрибутивами*, например Ubuntu, Red Hat, openSUSE), которые включают инструкции по установке, программное обеспечение и поддержку.

GPL

Linux и программное обеспечение GNU распространяются согласно условиям общедоступной лицензии GNU General Public License (GPL, www.gnu.org/licenses/licenses.html). В GPL говорится, что вы имеете право копирования, модификации и дальнейшего распространения кода оговариваемого в соглашении. Но при дальнейшем распространении кода вы должны также распространять с ним такую же лицензию, не отделяя таким образом код и лицензию друг от друга. Если вы получаете исходный код через Интернет, то, соблюдая условия распространения программ с лицензией GPL, в случае последующей модификации этого кода и последующего распространения исполняемой версии программы вы должны также распространять и модифицированный исходный код, и GPL-соглашение. Поскольку эти меры являются полной противоположностью порядка соблюдения обычных авторских прав — copyright (они предоставляют, а не ограничивают права), их назвали copyleft. (Данный абзац не является официальным изложением GPL, его задача состоит лишь в том, чтобы дать вам представление о работе лицензии. Когда потребуется сама лицензия GPL, нужно будет обратиться к ее официальному тексту.)

Получайте удовольствие!

Два ключевых слова для Linux — «Получайте удовольствие!» («Have Fun!»). Эти слова появляются в приглашениях и документации. Развитие UNIX, а теперь и Linux просто наполнено юмором, который встречается в системе повсюду. Например, слово меньше (less) используется вместо слова больше (more): в GNU утилита разбиения на страницы, которая в UNIX называлась more, заменена усовершенствованной утилитой less. Утилита для просмотра документов PostScript названа ghostscript (призрачным сценарием), и один из нескольких вариантов замены редактора vi назван elvis. По ана-

логии с тем, что компьютеры, оборудованные процессорами Intel, имеют снаружи логотип «Intel Inside» (Интел внутри), некоторые Linux-машины оживляются логотипами «Linux Inside». Да и сам Торвальдс был замечен в футболке с логотипом «Linus Inside».

Что можно отнести к достоинствам Linux?

В последние годы Linux позиционируется в качестве мощной и инновационной UNIX-подобной системы. Ее популярность уже превзошла популярность ее UNIX-предшественников. Хотя она во многом подражает UNIX, операционная система Linux отошла от UNIX по ряду существенных направлений: ядро Linux разработано независимо от BSD и System V, продолжающаяся разработка Linux осуществляется объединенными усилиями многих талантливых людей по всему миру, и Linux сделал мощность UNIX доступной кругу как деловых людей, так и простых компьютерных пользователей. Используя Интернет, современные опытные программисты отправляют дополнения и улучшения Линусу Торвальдсу, кураторам проекта GNU или одному из других авторов Linux.

Стандарты В 1985 году представители компаний всей компьютерной отрасли объединились в целях разработки стандарта POSIX (Portable Operating System Interface for Computer Environments) — интерфейса переносимой операционной системы, который был основан большей частью на определении интерфейса UNIX System V Interface Definition (SVID) и на других более ранних попытках стандартизации. Эти усилия были простимулированы правительством США, которое нуждалось в стандартной вычислительной среде, чтобы минимизировать свои затраты на обучение и приобретение оборудования. Выпущенный в 1988 году, POSIX представлял собой сборник IEEE-стандартов, определяющих требования к интерфейсу прикладных программ — API (application program interface), оболочке и интерфейсу утилит операционной системы. Хотя нацеленность этого сборника была на UNIX-подобные системы, стандарты могли применяться для любых совместимых операционных систем. Теперь, когда эти стандарты получили признание, у разработчиков программного обеспечения появилась возможность разрабатывать приложения, которые работают на всех, соответствующих этим стандартам версиях UNIX, Linux и других операционных систем.

Приложения Для Linux доступен огромный выбор приложений, как бесплатных, так и коммерческих, а также широкое разнообразие инструментальных средств для обработки графики и текста, решения сетевых задач, обеспечения безопасности, администрирования системы, работы в качестве веб-сервера и для применения во многих других сферах. В последнее время крупные компании по разработке программного обеспечения увидели преимущества в поддержке Linux и теперь содержат штат программистов, занимающихся конструированием и программированием ядра Linux, GNU, KDE и другого программного обеспечения, которое работает под управлением Linux. К примеру, основной организацией, поддерживающей Linux, является компания IBM (www.ibm.com/linux). Linux все больше соответствует стандартам POSIX, а некоторые дистрибутивы и составные части других операционных систем отвечают им в полной мере. Это свидетельствует о том, что Linux становится широко распростра-

ненной операционной системой и рассматривается в качестве привлекательной альтернативы другим популярным операционным системам.

Внешние устройства Другим привлекательным для пользователей аспектом Linux является огромное количество поддерживаемых этой системой внешних устройств и та скорость, с которой появляется поддержка новых внешних устройств. Зачастую Linux опережает своих конкурентов в поддержке внешних устройств или интерфейсных карт. К сожалению, имеется также и ряд неподдерживаемых внешних устройств, в частности графических карт, и причина здесь в том, что производители своевременно не выпускают спецификации или исходный код для драйверов или вообще не делают этого.

Программное обеспечение Для пользователей также представляет интерес объем доступного программного обеспечения не только в виде исходного кода (нуждающегося в компиляции), но также в виде готового двоичного кода, который легко устанавливается и запускается. Эти программы включают не только свободное программное обеспечение. Netscape, к примеру, был доступен для Linux с самого начала и включал поддержку языка Java еще до того, как она стала доступна в поставках многих коммерческих распространителей. В этой системе также доступны и родственные ему браузеры, почтовые клиенты и программы для чтения новостей, а также программы, выполняющие многие другие функции программ Mozilla-Thunderbird-Firefox.

Платформы Linux предназначена не только для платформ на базе процессоров Intel (в число которых теперь входят и компьютеры Apple): она была перенесена и работает под управлением Power PC, включая более старые компьютеры компании Apple (ppclinux), компьютеры компании Compaq (урожденная Digital Equipment Corporation) на основе процессоров Alpha, машины на базе MIPS-процессоров, компьютеры с процессорами Motorola 68K, различные 64-разрядные системы и машины компании IBM S/390. Но Linux работает не только на однопроцессорных машинах: начиная с версии 2.0, она работает и на многопроцессорных машинах (SMP). В нее также включен планировщик O(1), существенно повышающий масштабируемость SMP-систем.

Эмуляторы Linux поддерживает программы, которые называются эмуляторами и запускают код, предназначенный для других операционных систем. За счет использования эмуляторов, под Linux могут запускаться некоторые программы, предназначенные для DOS, Windows и Macintosh. Например, Wine (www.winehq.com) представляет собой реализацию Windows API, имеющую открытый код и запускающуюся в виде надстройки над X Window System и UNIX/Linux.

Виртуальные машины Виртуальная машина (VM, или гость) предстает перед пользователем и запускаемым программным обеспечением в виде полноценной физической машины. Но тем не менее это лишь одна из потенциально возможного множества VM, запущенных на одной и той же физической машине (хозяине, или хосте). Программное обеспечение, предоставляющее виртуализацию, называется монитором виртуальной машины (VMM) или гипервизором. На каждой виртуальной машине может быть запущена операционная система, отличная от тех, что запущены на других виртуальных машинах. Например, на одном и том же хосте можно создать виртуаль-

ные машины, на которых запущены Windows, Ubuntu 7.10, Ubuntu 9.04 и Fedora 10. Многозадачные операционные системы позволяют запускать множество программ на одной и той же физической системе. Точно так же гипервизор позволяет запускать множество операционных систем (виртуальных машин) на одной и той же физической системе.

Виртуальные машины предоставляют множество преимуществ по сравнению с отдельными, специализированными машинами:

- **Автономность** — каждая виртуальная машина изолирована от других виртуальных машин, запущенных на том же хосте: поэтому отказ или опасная ситуация на одной виртуальной машине не оказывает никакого влияния на другие виртуальные машины.
- **Безопасность** — при взломе единой серверной системы, на которой запущено сразу несколько серверов, оказываются взломанными все эти серверы. А если каждый сервер запущен на своей собственной виртуальной машине, то утечка информации распространяется только на взломанный сервер, все другие серверы остаются в безопасном состоянии.
- **Расход энергии** — при использовании виртуальных машин одна мощная машина может заменить множество менее мощных, сокращая тем самым расход электроэнергии.
- **Разработка и поддержка** — множество виртуальных машин, на каждой из которых запущены различные версии операционной системы и (или) другие операционные системы, могут способствовать разработке и поддержке программного обеспечения, рассчитанного на работу сразу в нескольких средах. При такой организации можно будет без труда протестировать продукт перед его выпуском в различных средах. Точно так же, когда пользователь отправляет сообщение об ошибке, эту ошибку можно воспроизвести на том же оборудовании, на котором она возникла.
- **Серверы** — в некоторых случаях разные серверы требуют разных версий системных библиотек. Тогда можно запустить каждый сервер на своей собственной виртуальной машине и сделать все это на одном и том же компьютерном оборудовании.
- **Тестирование** — при использовании виртуальных машин можно проводить эксперименты с самыми передовыми операционными системами и приложениями, не опасаясь за работу базовой (стабильной) системы и делать все это на одной и той же машине.
- **Сети** — можно устанавливать и тестировать на одной машине сети, состоящие из нескольких систем.
- **Песочницы** — виртуальная машина предоставляет возможность создания так называемой песочницы — области (системы), в которой можно работать, не беспокоясь о результатах своей работы или о необходимости привести все в исходное состояние.
- **Фиксирование мгновенных состояний** — можно зафиксировать мгновенное состояние виртуальной машины и вернуть ее в то состояние, в котором она была на момент фиксации, путем простой повторной загрузки виртуальной машины данными этого зафиксированного состояния.

Xen Программа Xen, созданная в Кембриджском университете и разрабатываемая в настоящее время сообществом по созданию программ с открытым кодом, является монитором виртуальной машины (VMM), имеющим открытый код. Использование Xen приводит к минимальным издержкам производительности по сравнению с исходной работой каждой из операционных систем.

В данной книге не даются описания установки или использования Xen. Дополнительный сведения о Xen доступны на главной странице посвященного этому монитору веб-сайта www.cl.cam.ac.uk/research/srg/netos/xen, а также на веб-сайте wiki.xensource.com/xenwiki.

VMware Компания VMware, Inc. (www.vmware.com) предлагает VMware Server, бесплатный, свободно загружаемый собственный продукт, который можно установить и запустить под Linux как простое приложение. VMware Server позволяет установить несколько виртуальных машин, в каждой из которых могут запускаться различные операционные системы, включая Windows и Linux. VMware также предлагает бесплатную программу VMware player, позволяющую запускать созданные виртуальные машины с сервером VMware Server.

KVM Виртуальная машина, основанная на использовании ядра (Kernel-based Virtual Machine, KVM; kvm.qumranet.com и libvirt.org), представляет собой виртуальную машину с открытым кодом, которая запускается как часть ядра Linux. Она работает только на системах с центральными процессорами Intel VT (VMX) или на системах с центральными процессорами AMD SVM.

Qemu Программа Qemu (bellard.org/qemu), созданная Фабрисом Белларом (Fabrice Bellard), представляет собой монитор VMM с открытым кодом, запускающийся как обычное приложение, не предъявляющее каких-либо требований к используемому центральному процессору. Она может запускать код, написанный для центральных процессоров, отличающихся от того, на котором работает хост-машина.

VirtualBox Программа VirtualBox (www.virtualbox.org) является виртуальной машиной с открытым кодом, разработанной компанией Sun Microsystems.

Почему Linux популярен среди компаний, производящих компьютерное оборудование, и среди разработчиков

Основа растущей популярности UNIX и Linux определяется двумя тенденциями развития компьютерной индустрии. Во-первых, технологические достижения в сфере компьютерного оборудования определяют востребованность операционной системы, способной реализовать все преимущества доступных компьютерных мощностей. В середине 70-х годов прошлого века мини-компьютеры стали выглядеть перспективнее больших универсальных компьютеров, поскольку в решении многих прикладных задач они могли выполнять те же самые функции с меньшими затратами. В последнее время мощные 64-разрядные микропроцессоры, устройства памяти, обладающие большой емкостью и невысокой ценой, и недорогие жесткие диски позволяют компаниям-производителям компьютерного оборудования устанавливать многопользовательские операционные системы на настольные компьютеры.

Специализированные операционные системы

Во-вторых, при постоянном снижении цен на компьютерное оборудование, его производители не могут больше вести разработку и поддержку *специализированных* операционных систем, то есть таких систем, которые созданы производителем компьютерного оборудования и являются его собственностью (например, компания DEC/Compaq является владельцем операционной системы VMS). Современные производители нуждаются в универсальной операционной системе, которую они могли бы легко адаптировать под свои машины.

Универсальные операционные системы

Универсальные операционные системы создаются вне стен компаний-производителей компьютерного оборудования и продаются (UNIX, OS X, Windows) или передаются (Linux) производителям. Linux является универсальной операционной системой потому что она запускается на различных типах оборудования, изготовленного разными производителями. Конечно, производителям выгоднее заплатить только за разработку и избежать выплат за каждый установленный экземпляр (что они вынуждены делать, оплачивая корпорации Microsoft каждую проданную ими копию Windows). В свою очередь, разработчики программного обеспечения должны снижать цены на свои продукты. Они не могут позволить себе создавать новые версии своих продуктов для работы под управлением множества специализированных операционных систем. И производители компьютерного оборудования, и разработчики программного обеспечения нуждаются в универсальной операционной системе.

Несмотря на то что система UNIX в свое время отвечала требованиям, предъявляемым компаниями-производителями компьютерного оборудования и исследователями к универсальным операционным системам, со временем она превратилась в более специализированную, по мере того как производители добавляли поддержку своих специализированных возможностей и внедряли новые программные библиотеки и утилиты. Система Linux появилась, чтобы удовлетворить обе потребности: быть универсальной операционной системой и обладать возможностью использования всего доступного оборудования.

Linux является переносимой системой

К *переносимым* относятся такие операционные системы, которые могут работать на множестве различных машин. Более 95 % всего объема операционной системы Linux написано на языке программирования C, и она обладает переносимостью, поскольку код написан на машинно-независимом языке высокого уровня. (Даже сам компилятор C написан на языке C.)

Поскольку операционная система Linux обладает переносимостью, она может быть адаптирована (перенесена) на различные машины и отвечать специальным требованиям. Например, Linux используется во встроенных компьютерах, являющихся составной частью мобильных телефонов, КПК, блоков кабельного телевидения, к которым подключается множество телевизоров. Ее файловая система способна использовать все преимущества высокоскоростных жестких дисков большой емкости. Не менее важно и то, что Linux изначально разрабатывалась как многопользовательская операционная система — она не подвергалась дополнительной переделке под обслуживание сразу нескольких клиентов. Распределение компьютерных ресурсов среди множества

пользователей и предоставление им возможности пользования общими данными и программами является ключевой особенностью этой системы.

Благодаря своим возможностям к адаптации и эффективному использованию доступного оборудования Linux работает не только на многих различных микропроцессорных системах, но и на больших универсальных машинах. Продвижению Linux способствует популярность оборудования, основанного на применении микропроцессоров; быстродействие этих микрокомпьютеров постоянно растет при практически неизменных ценах. Linux как нельзя лучше подошла для установки на быстродействующих микрокомпьютерах, позволив им заменить в качестве настольных систем рабочие станции.

Широкое признание пошло на пользу обеим категориям пользователей: тем, кто не любил изучать новую операционную систему, поставляемую с каждым новым оборудованием, и системным администраторам, которым нравилось работать в стабильном программном окружении.

Появление стандартной операционной системы подстегнуло разработчиков программного обеспечения. Теперь производители программных продуктов получили возможность создавать единую версию продукта, способную работать на машинах разных производителей.

Язык программирования C

Кен Томпсон (Ken Thompson) создал операционную систему UNIX в 1969 году на ассемблере машины PDP-7. Язык ассемблера является машинно-зависимым: программы, написанные на языке ассемблера, работают только на одной машине или, в лучшем случае, на одном семействе машин. Поэтому исходная версия операционной системы UNIX не могла быть просто перенесена для работы на других машинах (она не обладала переносимостью).

Чтобы сделать UNIX переносимой системой, Томпсон разработал язык программирования B, машинно-независимый язык, прототипом которого послужил язык BCPL. Деннис Ричи (Dennis Ritchie) разработал язык программирования C, модифицировав B, и вместе с Томпсоном в 1973 году переписал UNIX на C. Первоначально C рекламировался как «переносимый ассемблер». Переработанную операционную систему можно было уже значительно проще перенести для работы на других машинах.

Эта разработка стала отправной точкой для языка C. История его появления вскрывает ряд причин, позволивших ему стать мощным инструментальным средством. Язык C может использоваться для написания машинно-независимых программ. Разработчик переносимой программы может с легкостью переместить ее на любой компьютер, имеющий компилятор языка C. К тому же этот язык разработан для получения на выходе компилятора весьма эффективного кода.

С появлением языка C у программистов отпала необходимость обращаться к языку ассемблера для получения эффективно работающего кода (то есть достаточно быстродействующего, поскольку ассемблер всегда генерирует более эффективный код, чем язык высокого уровня).

Язык C хорошо подходит в качестве системного языка. На нем можно написать компилятор или операционную систему. Он обладает хорошо выраженной структурой,

но это не говорит о том, что он является типичным языком высокого уровня. Язык C позволяет программисту работать с битами и байтами, поскольку это крайне необходимо при написании операционной системы. В то же время у него есть конструкции высокого уровня, позволяющие заниматься эффективным модульным программированием.

В конце 80-х годов Американский национальный институт стандартов (American National Standards Institute, ANSI) определил стандарт языка C, который обычно упоминается как ANSI C или C89 (по году опубликования). Десять лет спустя вышел стандарт C99, который во многом поддерживается C-компилятором проекта GNU (который называется gcc). Исходная версия языка часто упоминается как Kernighan & Ritchie (или K&R) C, она названа по именам авторов книги (Кернигана и Ричи), в которой впервые был описан язык C.

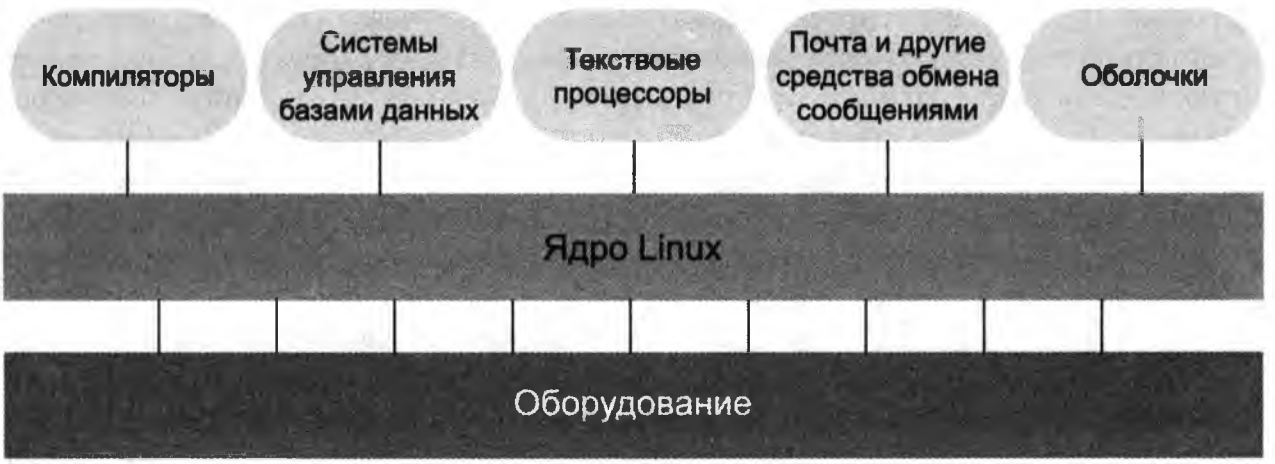

Рис. 1.1. Многоуровневое представление операционной системы Linux

Еще один исследователь, работающий в Bell Labs, Бьерн Страуструп (Bjarne Stroustrup), создал объектно-ориентированный язык программирования C++, который построен на основе языка C. Поскольку знание объектно-ориентированного программирования сегодня является требованием многих работодателей, C++ во многих случаях является более предпочтительным языком, чем C. Можно также остановить свой выбор на языке Objective-C, который использовался при написании первых веб-браузеров. C-компилятор, который используется в проекте GNU, поддерживает C, C++ и Objective-C.

Обзор Linux

Операционная система Linux обладает многими уникальными и мощными свойствами. Как и все другие операционные системы, она является управляющей программой для компьютеров. Но так же как и UNIX, она представляет собой хорошо продуманное семейство служебных программ (см. рис. 1.1) и набор инструментов, позволяющий пользователю подключать и использовать все эти утилиты для построения систем и приложений.

Linux обладает интерфейсом, программируемым на уровне ядра

Ядро Linux является сердцем этой операционной системы, оно отвечает за распределение компьютерных ресурсов и планирование выполнения пользовательских задач, при котором каждая из них получает свою справедливую долю системных ресурсов, включая доступ к центральному процессору, внешним устройствам, таким как жесткий диск, привод DVD и компакт-дисков, принтеры и накопители на магнитной ленте. Программы взаимодействуют с ядром посредством *системных вызовов*, специальных функций с хорошо известными именами. Программист может воспользоваться одним системным вызовом для взаимодействия со многими разновидностями устройств. Например, одним системным вызовом `write()` вместо множества вызовов, приспособленных под различные устройства. При выдаче программой системного запроса `write()` ядро интерпретирует его содержимое и передает запрос соответствующему устройству. Эта гибкость позволяет старым утилитам работать с устройствами, которых на момент создания этих утилит еще не было. Это также позволяет перемещать программы на новые версии операционной системы без необходимости их переписывать (если только новая версия умеет распознавать те же самые системные вызовы).

Linux может поддерживать работу множества пользователей

В зависимости от состава компьютерного оборудования и типа выполняемых компьютером задач система Linux способна поддерживать от одного до более чем тысячи пользователей, параллельно запускающих различные наборы программ. Для каждого отдельного пользователя компьютер, который может одновременно использоваться многими людьми, обходится дешевле компьютера, находящегося в индивидуальном пользовании. Снижение стоимости обусловлено тем, что один пользователь в принципе не может воспользоваться сразу всеми ресурсами, предлагаемыми компьютером. То есть никто не в состоянии постоянно задействовать все принтеры, всю системную память, заставить все диски считывать и записывать информацию, постоянно пользоваться Интернетом и занимать одновременно все терминалы. А многопользовательская операционная система позволяет многим людям практически одновременно использовать системные ресурсы. Коэффициент использования дорогостоящих устройств может быть повышен до максимума, а стоимость, приходящаяся на одного пользователя, снижена до минимума, — что и является главной целью создания многопользовательской операционной системы.

Linux может одновременно выполнять множество задач

Linux является полностью защищенной многозадачной операционной системой, позволяющей каждому пользователю одновременно запускать более одной задачи. Процессы могут обмениваться данными друг с другом, оставаясь полностью защищенными один от другого, точно так же, как ядро остается защищенным от всех процессов. Можно запускать несколько задач в фоновом режиме, уделяя все свое внимание

только той задаче, которая отображается на экране, и можно переключаться между задачами в обоих направлениях. При работе в системе X Window System появляется возможность запуска разных программ в разных окнах на одном и том же экране и вести наблюдение за работой всех этих программ. Эта возможность помогает повысить продуктивность работы пользователей.

Linux предоставляет безопасную иерархическую файловую систему

Файл представляет собой набор информации, например текст служебной записки или отчета, сводку о продаже товаров, изображение, песню или исполняемую программу. Каждый файл хранится под уникальным идентификатором на накопителе информации, например на жестком диске. Файловая система Linux представляет собой структуру, в которой файлы распределены по каталогам, напоминающим папки или ящички. В каждом каталоге могут храниться файлы и другие каталоги. В свою очередь, каталоги распределены по другим каталогам и т. д., составляя древовидную структуру. Эта структура помогает пользователям отслеживать большое количество файлов, группируя взаимосвязанные файлы в каталогах. У каждого пользователя есть один основной каталог и множество подкаталогов, соответствующих его потребностям (рис. 1.2).

Стандарты Вынашивая замысел облегчения жизни системным администраторам и разработчикам программного обеспечения, в Интернете собралась группа, разработавшая стандарт файловой системы — Linux Filesystem Standard (FSSTND), который с тех пор развился в стандарт иерархии файловой системы — Linux Filesystem Hierarchy Standard (FHS). До принятия данного стандарта ключевые программы в разных дистрибутивах Linux находились в разных местах. Сегодня можно, находясь в системе Linux, совершенно предсказуемо найти любую нужную стандартную программу в одном и том же месте.

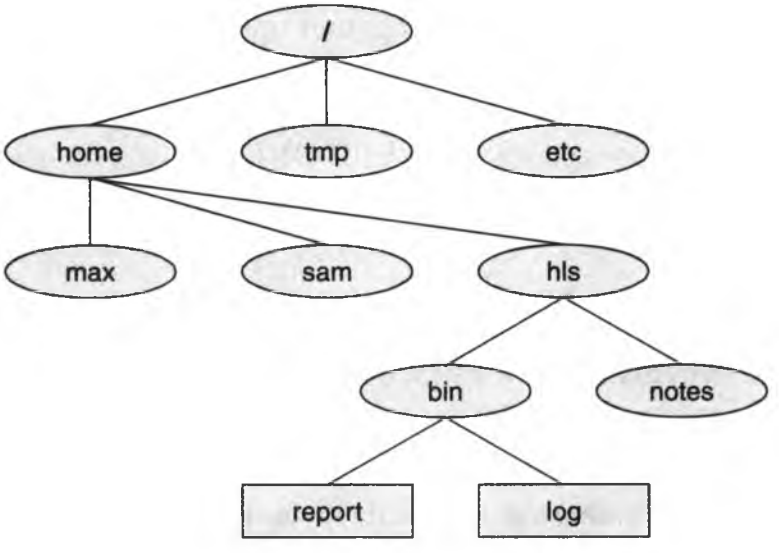

Рис. 1.2. Структура файловой системы Linux

Ссылки *Ссылка* позволяет указанному файлу иметь связь с двумя и более именами. Альтернативные имена могут находиться в одном каталоге с исходным файлом или в другом каталоге. Ссылки могут заставить один и тот же файл появляться сразу в нескольких пользовательских каталогах, позволяя этим пользователям легко организовать его совместное использование. В Windows для описания такой же возможности вместо *ссылки* используется термин *ярлык* (shortcut). Пользователям Macintosh более известен термин *псевдоним* (alias). В Linux *псевдоним* отличается от *ссылки*; он относится к свойствам макрокоманд, предоставляемых оболочкой.

Безопасность Как и многие другие многопользовательские операционные системы, Linux позволяет пользователям защищать их данные от доступа со стороны других пользователей. Она также позволяет совместно использовать избранные данные и программы посредством простой, но эффективной схемы защиты. Уровень безопасности предоставляется в виде прав доступа к файлу, которые ограничивают круг пользователей, имеющих право считывать данные из файла, записывать их в файл или запускать файл на выполнение. Позже в Linux были реализованы списки управления доступом — Access Control Lists (ACL), которые дают пользователям и администраторам более широкие возможности по управлению правами доступа к файлу.

Оболочка: интерпретатор команд и язык программирования

В текстовой среде оболочка — интерпретатор команд — работает как интерфейс между вами и операционной системой. Когда команда вводится и появляется на экране, оболочка интерпретирует эту команду и вызывает нужную вам программу. В Linux доступны сразу несколько оболочек. К наиболее популярным относятся:

- Оболочка Bourne Again Shell (bash), усовершенствованная версия исходной оболочки Bourne Shell (оригинальной оболочки UNIX).
- Оболочка Debian Almquist Shell (dash), сокращенная версия bash, обладающая меньшими возможностями. Для ускорения процесса загрузки большинство сценариев запуска оболочки вызывают dash вместо bash.
- Оболочка TC Shell (tcsh), усовершенствованная версия C Shell, оболочки, разработанной в качестве составной части BSD UNIX.
- Оболочка Z Shell (zsh), вобравшая в себя свойства сразу нескольких оболочек, включая Korn Shell.

Поскольку разные пользователи отдают предпочтения разным оболочкам, на многопользовательских системах могут одновременно использоваться сразу несколько разных оболочек. Возможность выбора оболочки демонстрирует еще одно преимущество операционной системы Linux: предоставление интерфейса, настроенного под каждого пользователя.

Сценарии оболочки Кроме осуществления своих функций по интерпретации команд, введенных с клавиатуры, и отправки этих команд операционной системе, оболочка является языком программирования высокого уровня. Команды оболочки могут быть организованы в файл, предназначенный для последующего использования (В Linux эти файлы называются *сценариями оболочки*; в Windows они называются

пакетными файлами). Подобная гибкость позволяет пользователям с относительной легкостью выполнять довольно сложные операции, зачастую путем выдачи коротких команд или создания, как ни странно, без особого труда, конкретных программ, выполняющих весьма сложные действия.

Образование имен файлов

Групповые символы и нестрогие ссылки на файлы

При вводе команды, обрабатываемой оболочкой, можно конструировать шаблоны, используя в них символы, имеющие для оболочки специальное значение. Эти символы называются *групповыми символами*. Шаблоны, которые называются *нестрогими ссылками на файлы*, представляют собой разновидность условного обозначения: вместо ввода полных имен файлов можно ввести шаблоны, а оболочка распространит их на соответствующие имена файлов. Нестрогие ссылки на файлы могут сократить время ввода длинных имен файлов или длинных наборов похожих имен. Например, оболочка может распространить шаблон `mak*` на файл по имени **make-3.80.tar.gz**. Шаблоны также могут пригодиться, если известна только часть имени файла или если трудно вспомнить точное написание этого имени.

Автозавершение

Совместно с библиотекой Readline, оболочка осуществляет автозавершение названий команд, имен файлов, путевых имен и имен переменных: вы набираете префикс и нажимаете клавишу ESCAPE, а оболочка выводит список элементов, начинающихся с этого префикса, или автоматически завершает имя элемента, если префикс указывает на уникальный элемент.

Ввод и вывод не привязаны к конкретному устройству

Перенаправление

Устройства (например, принтер или терминал) и дисковые файлы фигурируют в Linux-программах в виде файлов. При выдаче команды операционной системе Linux можно проинструктировать ее на отправку выводимой информации на любое из нескольких устройств или файлов. Такое переключение называется *перенаправлением* вывода.

Непривязанность к конкретному устройству

Точно так же ввод данных в программу, который обычно происходит с помощью клавиатуры, может быть перенаправлен на данные, поступающие из файла, хранящегося на диске. Ввод и вывод не привязаны к конкретному устройству, то есть они могут быть перенаправлены на получение данных или на их выдачу на любое соответствующее устройство.

К примеру, утилита **cat** обычно отображает содержимое файла на экране. При запуске команды `cat` можно без труда заставить ее отправить выводимую информацию не на экран, а в дисковый файл.

Функции оболочки

Одной из наиболее важных функций оболочки является возможность ее использования в качестве языка программирования. Поскольку оболочка является интерпретатором, она не компилирует написанные для нее программы, а интерпретирует эти программы при каждой их загрузке с диска. Загрузка и интерпретация программ может занимать значительное время.

Многие оболочки, включая Bourne Again Shell, поддерживают свои функции, сохраняя их в памяти, что исключает необходимость чтения их с диска при каждом выполнении. Оболочка также сохраняет функции во внутреннем формате, что не требует тратить слишком много времени на их интерпретацию.

Управление заданиями

Управление заданиями является функцией оболочки, позволяющей пользователям работать сразу с несколькими заданиями и переключаться между ними в любом желаемом направлении. При запуске задания оно зачастую выполняется в активном режиме и поэтому подключается к терминалу. Используя управление заданиями, можно перевести текущее задание в фоновый режим и продолжить его выполнение в этом режиме, работая тем временем над другим заданием или просматривая ход выполнения другого задания в активном режиме. Если задание, выполняемое в фоновом режиме, станет нуждаться в вашем внимании, его можно перевести в активный режим, и оно опять подключится к терминалу. (Концепция управления заданиями берет свое начало в BSD UNIX, где такое управление впервые появилось в оболочке C Shell.)

Богатая коллекция полезных утилит

В Linux включено целое семейство, состоящее из нескольких сотен полезных программ, зачастую представленных нам в виде команд. Эти полезные программы, или утилиты, выполняют функции, постоянно востребуемые пользователями. Утилита sort, к примеру, выстраивает списки (или группы списков) в алфавитном или числовом порядке и может быть использована для сортировки списков по номерам деталей, по фамилиям, городам, почтовым индексам, телефонным номерам, возрасту, размеру, цене и т. д. Утилита sort является важным инструментом программирования, составляющим часть стандартной Linux-системы. Другие утилиты позволяют пользователю создавать, отображать, выводить на печать, копировать, удалять, редактировать, форматировать файлы, осуществлять в них поиск, а также редактировать и набирать текст. Утилита man (от слова manual — руководство) и утилита info предоставляют интерактивную документацию по Linux.

Взаимодействие процессов

Каналы и фильтры

Linux позволяет пользователям устанавливать в командной строке каналы и фильтры. Канал отправляет вывод одной программы в качестве входных данных другой программе. Фильтр является специфическим видом канала, который обрабатывает поток данных на входе, и формирует поток данных на выходе. Фильтр обрабатывает вывод другой программы, выдавая измененный результат. Затем выходная информация фильтра становится вводом для другой программы.

Каналы и фильтры часто объединяют утилиты для выполнения той или иной специфической задачи. Например, можно воспользоваться каналом для отправки выходных данных утилиты sort утилите head (фильтру, выводящему первые десять строк введенной в него информации); затем можно воспользоваться другим каналом для отправки вывода третьей утилите — lpr, которая отправляет данные на принтер.

Таким образом, в одной командной строке можно использовать сразу три утилиты для сортировки и распечатки части файла.

Администрирование системы

В Linux системный администратор зачастую является владельцем и единственным пользователем системы. Этот человек несет большую ответственность. В первую очередь это может касаться настройки системы, установки программного обеспечения и возможного редактирования конфигурационных файлов. После установки и запуска системы системный администратор отвечает за загрузку и установку программного обеспечения (включая обновление операционной системы), создание резервных копий и восстановление файлов, а также за обслуживание таких системных объектов, как принтеры, терминалы, серверы и локальные сети. Системный администратор несет также ответственность за настройки учетных записей для новых пользователей многопользовательской системы, за ввод системы в рабочее состояние и вывод ее из него по мере необходимости, за отслеживание состояния системы и за решение всех возникающих проблем.

Дополнительные свойства Linux

Разработчики Linux включили в свою операционную систему свойства из BSD, System V и Solaris (ОС компании Sun Microsystems), а также ряд совершенно новых свойств. Хотя большинство инструментальных средств, присущих UNIX, существует и в Linux, в некоторых случаях эти средства были заменены их более современными аналогами. В этом разделе дается описание некоторых популярных инструментальных средств и функций, доступных в системе Linux.

Графические пользовательские интерфейсы — GUI

X11 Система X Window System (также называемая X или X11) была частично разработана исследователями Массачусетского технологического института (Massachusetts Institute of Technology, MIT) и послужила основой для графических пользовательских интерфейсов Linux. Используя экран терминала или рабочей станции, который поддерживает X, пользователь может взаимодействовать с компьютером посредством множества окон, расположенных на экране, посредством отображения графической информации или за счет использования приложений, специально предназначенных для рисования изображений, отслеживания процессов или просмотра отформатированных выходных данных. Система X — это протокол передачи данных по сети, позволяющий пользователю открыть окно на рабочей станции или компьютерной системе, которое дистанционно управляется тем центральным процессором, который сгенерировал это окно.

Aqua Mac OS X была выпущена с двумя графическими интерфейсами, которые могут использоваться одновременно. Пользователям большинства машин Macintosh хорошо известна система Aqua, представляющая собой стандартный графический интерфейс Mac OS X. Aqua основана на технологии изображе-

ния под названием Quartz и имеет для приложений стандартный внешний вид и характер поведения. Кроме этого Mac OS X поддерживает X11, в которой также используется Quartz.

Менеджер рабочего стола

Обычно в качестве надстройки над системой X запускаются еще два уровня: менеджер рабочего стола (desktop manager) и оконный менеджер (window manager). Менеджер рабочего стола — это ориентированный на применение изображений пользовательский интерфейс, позволяющий взаимодействовать с системными программами за счет работы со значками, а не с набором соответствующих команд в оболочке. На большинстве дистрибутивов Linux по умолчанию работает менеджер рабочего стола GNOME (www.gnome.org), но на этих дистрибутивах может также работать KDE (www.kde.org) и ряд других менеджеров рабочего стола. Система Mac OS X управляет рабочим столом в Aqua, но не в X11, то есть менеджер рабочего стола при работе под X11 отсутствует.

Оконный менеджер

Оконным менеджером называется программа, которая работает под управлением менеджера рабочего стола и позволяет вам открывать и закрывать окна, запускать программы и настраивать мышь на разные действия в зависимости от того, как и где произошел щелчок. Кроме этого оконный менеджер придает экрану индивидуальный внешний вид. Microsoft Windows позволяет изменять цвет ключевых элементов окна, а оконный менеджер под управлением системы X позволяет настраивать под свои нужды внешний вид и поведение всего экрана: можно изменить внешний вид и особенности работы окна (задавая ему разные границы, кнопки и полосы прокрутки), установить виртуальные рабочие столы, создать меню и сделать еще многое другое.

Под X и Linux могут работать несколько популярных оконных менеджеров. В большинстве дистрибутивов Linux используются менеджеры Metacity (по умолчанию работающий под GNOME) и kwin (по умолчанию работающий под KDE). Доступны также другие оконные менеджеры, например Sawfish и WindowMaker.

Под управлением Mac OS X многие окна управляются на уровне менеджера Quartz, применяемого для формирования внешнего вида и поведения Apple Aqua. Исключительно для приложений X11 эта задача выполняется менеджером quartz-wm, который подражает внешнему виду и поведению Apple Aqua, поэтому приложения X11 на рабочем столе Mac имеют такой же внешний вид, как и собственные приложения Mac OS X.

Сетевые и межсетевые утилиты

Сетевая поддержка в Linux включает множество утилит, обеспечивающих доступ к удаленным системам через разнообразные сети. Кроме отправки электронной почты пользователям, работающим на других системах, можно получать доступ к файлам на дисках, смонтированных на других компьютерах, как будто они установлены локально, и точно так же обеспечивать доступность своих файлов другим системам. Можно копировать файлы в обе стороны, запускать программы на удаленных системах, отображая результаты на локальной системе, и выполнять множество других операций по локальным (local area network, LAN) и глобальным сетям (wide area network, WAN), включая Интернет.

Над этим сетевым доступом надстраивается уровень, имеющий широкий диапазон прикладных программ, которые распространяют ресурсы компьютера на просторы всего земного шара. Вы можете связываться с людьми по всему миру, собирать информацию по широкому спектру тем, быстро и надежно загружать новое программное обеспечение через Интернет.

Разработка программного обеспечения

Одним из самых впечатляющих достоинств Linux является богатый выбор средств и методов разработки программного обеспечения. Linux поддерживает компиляторы и интерпретаторы для многих компьютерных языков. Кроме C и C++, в число языков, доступных в Linux, входят Ada, Fortran, Java, Lisp, Pascal, Perl и Python. Утилита bison генерирует код синтаксического анализатора (парсера), который упрощает написание программ, предназначенных для компиляции (эта утилита относится к инструментам, которые проводят синтаксический разбор файлов, содержащих структурированную информацию). Утилита flex генерирует сканеры (код, распознающий лексические структуры в тексте). Утилита make, а также система конфигурации и компоновки, имеющаяся в GNU (GNU Configure and Build System), облегчают управление сложными проектами по разработке программного обеспечения. Системы управления исходным кодом (Source code management systems), например CVS, упрощают управление версиями программного продукта. Ряд отладчиков, к числу которых относятся ups и gdb, помогает отследить и устранить дефекты программного обеспечения. GNU C-компилятор (gcc) работает совместно с утилитой-профилятором gprof, которая помогает программистам выявить потенциально узкие места в производительности создаваемых программ. В компиляторе C используются ключи для осуществления всесторонней проверки C-кода, повышая тем самым переносимость кода и сокращая время, необходимое на отладку программы.

Среда разработки Apple Xcode, работающая под управлением OS X, обеспечивает унифицированный графический внешний интерфейс для большинства всех этих инструментов, а также другие возможности и свойства.

Заключение

Операционная система Linux прошла путь развития от наследницы UNIX до популярной альтернативы традиционным системам (таким как Windows), доступной для работы на микрокомпьютерном оборудовании (на персональных компьютерах). Пользователи UNIX найдут в Linux знакомую им среду окружения. Дистрибутивы Linux дополнены востребованными UNIX-утилитами, предоставленными программистами со всего мира, включая набор инструментов, разработанных как часть проекта GNU. Сообщество Linux не прекращает разработку этой системы. Поддержка новых микрокомпьютерных устройств и их новых свойств добавляется практически сразу же после того, как это оборудование становится доступным, а имеющиеся в Linux инструментальные средства продолжают совершенствоваться. Благодаря многим коммерческим пакетам программного обеспечения, которые могут работать на платформах Linux, и многим производителям компьютерного оборудования, пред-

лагающим использовать Linux на своих системах, вполне очевидно, что эта система получила мощное развитие от студенческого проекта и до операционной системы, которую выбирают для научного, учебного, промышленного, профессионального и личного пользования.

Упражнения

1. Что такое свободное программное обеспечение (free software)? Перечислите три его признака.
2. Чем обусловлена популярность Linux? Почему эта операционная система популярна в учебных и научных кругах?
3. Что такое многопользовательские системы? Почему они пользуются успехом?
4. Что такое фонд свободного программного обеспечения (Free Software Foundation) и что такое GNU? Что такое Linux? Какие составные части операционной системы Linux предоставлены каждой из этих организаций? Кто еще помог в создании и совершенствовании этой операционной системы?
5. На каком языке написана Linux? Каким образом этот язык повлиял на успех Linux?
6. Что такое служебная программа (утилита)?
7. Что такое оболочка? Как она взаимодействует с ядром? С пользователем?
8. Как можно использовать служебные программы и оболочку для создания своих собственных приложений?
9. Почему файловая система Linux считается *иерархической*?
10. В чем разница между мультипроцессором и мультипроцессорной системой?
11. Приведите пример востребованности мультипроцессорной системы.
12. Каково приблизительное количество людей, создававших Linux? В чем уникальность этого проекта?
13. Назовите ключевые понятия общедоступной лицензии GNU General Public License.

ЧАСТЬ I

Операционные системы Linux и Mac OS X

Глава 2. Начнем 42
Глава 3. Утилиты 65
Глава 4. Файловая система 97
Глава 5. Оболочка 137

Глава 2. Начнем!

Итак, вы, наверное, уже сидите перед монитором, подключенным к компьютеру, на котором запущена система Linux. Возможно, вы работаете с графическим пользовательским интерфейсом (GUI) или с текстовым интерфейсом. Эта книга посвящена текстовому интерфейсу, который также называется интерфейсом командной строки (command-line interface, CLI). Если вы работаете с GUI, то для изучения всех примеров, приводимых в данной книге, нужно воспользоваться эмулятором терминала, например xterm, Konsole, GNOME Terminal, Terminal (под управлением Mac OS X), или виртуальной консолью (см. раздел «Использование виртуальных консолей» на с. 61).

Эта глава начинается с рассмотрения типографских соглашений, используемых в данной книге, затем следует раздел, посвященный входу в систему. В следующем за ним разделе представляется оболочка и даются объяснения о том, как можно справиться с ошибками в командной строке. Затем следует краткое напоминание о правах, связанных с работой в режиме привилегированного пользователя (root privileges), и даются советы о том, как не допускать ошибок, вводящих систему в нерабочий режим или в режим, затрудняющий ее работу. После этого приводится подборка источников дополнительной информации о Linux. В заключение приводится дополнительная информация о входе в систему, включая способы изменения пароля.

Внимательно прочитайте следующее далее предупреждение, касающееся опасности неправильного использования прав привилегированного пользователя (root-прав). С учетом данного предупреждения, можно вполне свободно экспериментировать с системой: вводить команды, создавать файлы, выполнять примеры, приводимые в данной книге, и получать удовольствие от работы с ней.

Соглашения, используемые в этой книге

В данной книге используются соглашения, позволяющие сделать приводимые объяснения короче и понятнее. Эти соглашения приводятся в нескольких следующих абзацах.

Mac OS X Термин Mac OS X относится как к Mac OS X, так и к Mac OS X Server. Важные различия между этими двумя системами в данной книге оговариваются особо.

Версии Mac OS X При ссылках на Mac OS X имеется в виду версия 10.5 (Leopard). Поскольку основное внимание в книге уделяется базовой операционной системе, которая от выпуска к выпуску новых подверсий OS X изменялась весьма незначительно, текст не утратит своей актуальности и для нескольких будущих выпусков. Все соответствующие коррективы и обновления можно найти на авторском веб-сайте (www.sobell.com).

Текст и примеры Текст приводится в данном формате, а примеры показываются с использованием моноширинного шрифта:

```
$ cat practice
Это небольшой файл, который я создал
с помощью текстового редактора.
```

Вводимые вами элементы

Все, что вы вводите с клавиатуры, будет показано с использованием `этого шрифта` в тексте и **этого шрифта** в примерах и копиях экрана. В предыдущем примере знак доллара (`$`) в первой строке является приглашением на ввод команды, отображаемым системой Linux, поэтому он не выделен жирным шрифтом, все остальное содержимое первой строки введено пользователем, поэтому оно имеет такое выделение.

Имена утилит

Для имен утилит используется шрифт Arial. В данной книге приводятся ссылки на текстовый редактор **emacs** и на утилиту **ls** или команду **ls** (или просто на ls), но указание по вводу в командную строку — `ls -a`, дается уже в другом шрифте. Таким образом в тексте обозначается различие между утилитами, которые являются программами, и инструкциями, которые вводятся в командную строку для вызова утилит.

Имена файлов

Имена файлов приводятся в книге **этим шрифтом**. Например, **memo5**, **letter.1283** и **reports**. Но имена файлов могут включать в себя буквы как в верхнем, так и в нижнем регистре. Популярные файловые системы Linux — ext2, ext3 и ext4, обладают чувствительностью к регистру букв, поэтому имена **memo5**, **MEMO5** и **Memo5** принадлежат трем разным файлам.

Файловая система HFS+, используемая в Mac OS X по умолчанию, не чувствительна к регистру букв, поэтому при использовании OS X имена **memo5**, **MEMO5** и **Memo5** будут указывать на один и тот же файл. Дополнительная информация изложена в разделе «Чувствительность к регистру букв» на с. 100.

Символьные строки

В тексте книги символы и символьные строки выделены `этим шрифтом`. Такое соглашение избавляет от необходимости заключения их в кавычки или другие разделительные знаки, ставящиеся до и после строки. В качестве примера посмотрите на следующую строку, отображаемую утилитой **passwd**: `Sorry, passwords do not match`[1].

Клавиши и символы

В данной книге ЭТОТ ШРИФТ используется для трех разновидностей элементов:

- Клавиш клавиатуры, например клавиш ПРОБЕЛ и ВВОД[2], ESCAPE и TAB.
- Символов, генерируемых клавишами, например для ПРОБЕЛов, генерируемых клавишей ПРОБЕЛ.
- Клавиш клавиатуры, которые нажимаются вместе с клавишей CONTROL, например CONTROL+D. (То, что буква `D` показана в верхнем регистре, не означает, что вам нужно нажимать еще и клавишу SHIFT; ввод сочетания CONTROL+D осуществляется удерживанием в нажатом состоянии клавиши CONTROL и нажатием клавиши d.)

---

[1] Для перевода строк сообщений об ошибках будет использоваться курсив (а во врезках — текст в скобках). — *Примеч. перев.*

[2] На разных клавиатурах используются разные клавиши перемещения курсора на начало следующей строки. В данной книге клавиша завершения строки всегда будет обозначаться как клавиша ВВОД. На вашей клавиатуре могут встречаться надписи RET, NEWLINE, ENTER, RETURN и т. д. Когда в материале книги встречается просьба нажать клавишу ВВОД, вам следует использовать клавишу, соответствующую вашей клавиатуре.

Приглашения и нажатие клавиши ВВОД

Большинство примеров включают в себя приглашения оболочки — признак того, что Linux ожидает ввода команды. Это приглашение может выводится в виде знака доллара ($), знака решетки (#), а в некоторых случаях и знака процента (%). Приглашения в данной книге не показываются жирным шрифтом, поскольку вы их не вводите. При проведении экспериментов с приводимыми в книге примерами не вводите знак приглашения. Если его ввести, пример работать не будет.

В примерах не обозначается нажатие клавиши ВВОД, необходимое для их выполнения. Рассмотрим следующий пример командной строки:

```
$ vim memo.1204
```

Чтобы воспользоваться данным примером в качестве модели для запуска текстового редактора vim, нужно ввести команду `vim memo.1204` и нажать клавишу ВВОД. (Для выхода из vim следует нажать клавиши ESCAPE ZZ; руководство по редактору vim приведено в главе 6.) Такой способ обозначения ввода команд позволяет приводить примеры в книге в том виде, в котором они появляются на экране.

Определения

Все словарные записи, имеющие метку FOLDOC, являются данью уважения Денису Хоу (Denis Howe), редактору бесплатного интерактивного словаря компьютерных терминов Free On-Line Dictionary of Computing (foldoc.org), и используются с его разрешения. Работа над этим сайтом, включающем определения, анекдоты и разные мелочи, продолжается.

Отрывки, помеченные как дополнительные, появляются на сером фоне. Этот материал содержит второстепенные сведения, представленные в главе, но часто включает в себя и довольно спорные концепции. При чтении главы лучше всего пропустить дополнительные разделы, чтобы вернуться к ним, когда уже усвоены основные, представленные в ней идеи. Читать эти абзацы необязательно.

URL-адреса (веб-адреса)

Веб-адреса, или URL-адреса, имеют подразумеваемый префикс http://, если только не показан другой префикс: ftp:// или https://. В браузерах, как правило, не нужно указывать префикс, если в его качестве должен быть http://, но при указании FTP-сервера или защищенного HTTP-сайта следует указывать и их префиксы. Таким образом, в браузере нужно указывать URL-адрес точно так же, как он показан в данной книге.

Врезки с советами, предупреждениями и заметками по безопасности

Следующие врезки выделяют ту информацию, которая может быть полезной при использовании или администрировании Linux-систем.

Может помочь избежать повторения типовых ошибок или может указать на дополнительную информацию.

Содержит предупреждение о каком-нибудь потенциальном подвохе.

Выделяет описание потенциальной проблемы безопасности. Обычно касается системных администраторов, но иногда и простых пользователей.

Вход в систему через терминал или эмулятор терминала

Перед выводом приглашения на вход в систему через терминал, эмулятор терминала или другое устройство, работающее в текстовом режиме, многие системы отображают сообщение, называемое *начальной заставкой* (issue) (которое хранится в файле /etc/issue). В этом сообщении обычно выводится название версии Linux, работающей в данной системе, имя системы и название устройства, в которое осуществляется вход. Простая заставка сообщает следующее:

```
Ubuntu 9.10 tiny tty1
```

За заставкой следует приглашение на вход. В ответ на приглашение системы следует ввести имя пользователя и пароль. В следующем примере показан пользователь Max, который входит в систему по имени tiny:

```
tiny login: max
Password:
Last login: Wed Mar 10 19:50:38 from dog
[max@tiny max]$
```

Если вы используете *терминал* и на экране не отображается приглашение login:, следует проверить подключение и питание терминала, а затем через некоторое время нажать клавишу ВВОД. Если приглашение login: не появилось и после этого, попробуйте нажать сочетание клавиш CONTROL+Q (подать сигнал на включение — Xon).

> **КТО В ПОСЛЕДНИЙ РАЗ ВХОДИЛ В СИСТЕМУ, ВЫ ИЛИ НЕ ВЫ?**
>
> При входе в среду, работающую в текстовом режиме, после ввода имени пользователя и пароля система отображает информацию о последнем входе в систему под этой учетной записью: когда и откуда был этот вход. Этой информацией можно воспользоваться, чтобы понять, что кто-то получил доступ к системе под вашей учетной записью уже после того, как вы пользовались ею в последний раз. Если такое действительно произошло, то, возможно, вы имеете дело с несанкционированным доступом со стороны пользователя, который узнал ваш пароль и вошел в систему под вашим именем. Для соблюдения мер безопасности поставьте в известность системного администратора о всем, что показалось вам подозрительным, и замените свой пароль.

Если вы работаете на компьютере Apple, PC, на другой Linux-системе или на рабочей станции, откройте программу, которая запускает безопасную утилиту ssh, небезопасную утилиту telnet или другое коммуникационное или эмуляционное программное обеспечение, которое используется вами для входа в систему, и предоставьте утилите имя или IP-адрес той системы, в которую желаете войти. При входе в систему убедитесь в том, что ввели имя пользователя и пароль именно так, как они были указаны при настройке учетной записи. Подпрограмма, проверяющая имя пользователя и пароль, чувствительна к регистру букв. Как и в большинстве других систем, в Linux пароль при вводе не отображается. По умолчанию система Mac OS X не разрешает удаленный вход в систему.

УТИЛИТА TELNET НЕ ОБЕСПЕЧИВАЕТ БЕЗОПАСНОСТЬ

Одна из причин небезопасности утилиты telnet заключается в том, что при входе в систему она отправляет ваше имя пользователя и пароль по сети открытым текстом, не препятствуя перехвату вашей информации по входу в систему и последующему входу в нее под вашей учетной записью. Утилита ssh шифрует всю информацию, отправляемую по сети, и если она доступна, лучше воспользоваться этой утилитой, а не telnet. Программа ssh была реализована на многих операционных системах, не только на Linux. Многие пользовательские интерфейсы к программе ssh включают в себя эмулятор терминала.

В следующем примере показан вход в систему, в котором ssh используется из системы Linux:

```
$ ssh max@tiny
max@tiny's password:
Permission denied, please try again.
          Доступ запрещен, пожалуйста, попробуйте еще раз
max@tiny's password:
Last login: Wed Mar 10 21:21:49 2005 from dog
[max@tiny max]$
```

В этом примере пользователь Max неправильно ввел свой пароль, получил сообщение об ошибке и еще одно приглашение и снова ввел пароль, на этот раз правильно. Если для системы, из которой вы выходите, используется то же самое имя пользователя, что и для системы, в которую вы входите, то его и следующий за ним знак at (@) можно пропустить. В данном примере Max должен был бы ввести команду `ssh tiny`.

После входа в систему появляется приглашение оболочки (или просто приглашение), свидетельствующее о том, что вход осуществлен успешно и система готова к вводу вашей команды. Первое приглашение оболочки может предваряться кратким сообщением, которое называется сообщением дня, или motd, и хранится в файле по имени /etc/motd.

Обычно в качестве приглашения используется знак доллара (`$`). Но не стоит волноваться, если у вас будет использоваться другой вид приглашения, так как примеры, приводимые в данной книге, будут работать независимо от того, как именно выглядит это приглашение. В предыдущем примере приветствие `$` (в последней строке) предваряется именем пользователя (`max`), знаком at (`@`), именем системы (`tiny`) и именем каталога, с которым работает пользователь Max. Информация о способах изменения приглашения изложена в главе 8, посвященной оболочке bash, и в главе 9, посвященной оболочке tcsh.

УБЕДИТЕСЬ В ПРАВИЛЬНОЙ УСТАНОВКЕ ЗНАЧЕНИЯ TERM

Переменная оболочки TERM устанавливает псевдографические символы на символьном терминале или эмуляторе терминала. Обычно значение TERM устанавливается без вашего участия, то есть вам не приходится делать это вручную. Если изображение на экране выглядит как то не так, нужно конкретно указать используемый терминал.

Работа с оболочкой

Перед включением графического пользовательского интерфейса (GUI) UNIX, а затем и Linux предоставляют текстовый интерфейс (который также называется интерфей-

сом командной строки — command-line interface, или CLI). В наши дни текстовый интерфейс доступен в том случае, когда вы входите в систему с терминала, эмулятора терминала или текстовой виртуальной консоли или же когда вы используете для входа в систему утилиту ssh или утилиту telnet.

При входе в систему и работе в текстовой (неграфической) среде и при использовании окна эмулятора терминала в графической среде в качестве интерпретатора команд используется оболочка. Она отображает приглашение, свидетельствующее о ее готовности к вводу вашей команды. В ответ на приглашение нужно ввести команду. Оболочка выполняет эту команду и отображает следующее приглашение.

Преимущества текстового интерфейса

Хотя сама идея использования текстового интерфейса может показаться устаревшей, этот интерфейс все же имеет право на существование и в современной обработке данных. В некоторых случаях администратор может пользоваться инструментарием командной строки либо из-за отсутствия его графического эквивалента, либо из-за того, что графический инструмент менее гибок или менее мощный, чем текстовый. Когда для входа в систему используется коммутируемая телефонная линия, использование GUI может оказаться неприемлемым из-за невысокой скорости подключения. Часто бывает, что графический интерфейс на серверной системе просто не установлен. Главной причиной такого исключения является то, что GUI потребляет слишком много системных ресурсов, а на сервере эти ресурсы лучше выделить для выполнения его основных задач. Кроме того, вопросы безопасности требуют, чтобы на серверной системе было запущено как можно меньше задач, поскольку каждая дополнительная задача повышает уровень уязвимости системы от внешних атак.

К тому же некоторые пользователи предпочитают текстовый интерфейс, поскольку он понятнее и эффективнее, чем GUI. Некоторые задачи, например смену прав доступа к файлам, проще решать из командной строки (см. описание утилиты chmod в части V).

Псевдографический интерфейс

Еще до появления графических интерфейсов изобретательные программисты создали текстовые интерфейсы, которые включали в себя такие графические элементы, как прямоугольники, рамки, оконтуренные примитивные окна, маркеры, а затем и цветовые выделения. Эти текстовые интерфейсы, называемые псевдографическими, позволили перекинуть своеобразный мост между текстовыми и графическими интерфейсами. Примером современной утилиты, использующей псевдографический интерфейс, может послужить утилита dpkg-reconfigure, которая реконфигурирует установленный пакет программ.

В этом разделе объясняется, как идентифицировать используемую вами оболочку, и описываются сочетания клавиш, используемые для исправления ошибок, допущенных в командной строке. В нем рассматриваются вопросы отмены выполнения запущенной команды и вкратце объясняются приемы редактирования содержимого командной строки. Работе с оболочками посвящены несколько следующих глав: представление оболочек приводится в главе 5, более подробное описание оболочки Bourne Again Shell, а также рассмотрение некоторых вопросов, касающихся оболочки TC Shell, дается в главе 8, глава 9 целиком посвящена оболочке TC Shell, а в главе 10 рассматриваются вопросы создания программ (сценариев оболочек) с использованием оболочки Bourne Again Shell.

С какой оболочкой вы работаете?

В данной книге рассматриваются оболочки Bourne Again Shell (bash) и TC Shell (tcsh). Скорее всего, у вас запущена оболочка bash, но, может быть, вы работаете с оболочкой tcsh или какой-нибудь другой оболочкой, например Z Shell (zsh). Идентифицировать работающую у вас оболочку можно с помощью утилиты ps. В ответ на приглашение оболочки наберите ps и нажмите клавишу ВВОД.

```
$ ps
  PID TTY          TIME CMD
 2402 pts/5    00:00:00 bash
 7174 pts/5    00:00:00 ps
```

В результате выполнения этой команды будет показано, что у вас запущены две утилиты или команды: bash и ps. Если вы работаете с tcsh, то вместо bash команда ps покажет tcsh. Если у вас запущена другая оболочка, то команда ps отобразит имя этой оболочки.

Исправление ошибок

В этом разделе объясняется, как нужно исправлять ошибки набора текста и другие ошибки, которые могут быть допущены во время вашей работы с текстовым дисплеем после входа в систему. Поскольку оболочка и большинство других утилит не осуществляют интерпретацию командной строки или другого текста до тех пор, пока не будет нажата клавиша ВВОД, то до ее нажатия все ошибки ввода можно легко исправить.

Делается это несколькими способами: проведением поочередного удаления символов, забоем сразу целого слова или забоем сразу всего текста командной строки до самого ее начала. Поле нажатия клавиши ВВОД исправлять ошибку будет уже поздно: нужно будет либо дождаться, пока команда доработает до конца, либо прервать ее выполнение.

Удаление символа

При вводе символов с клавиатуры вы можете стереть ошибку, нажимая клавишу стирания для каждого символа, который нужно удалить. Клавиша стирания может удалить требуемое количество символов. Но она не может удалить символы, находящиеся перед началом строки.

Клавишей стирания по умолчанию является забой — BACKSPACE. Если эта клавиша не работает, попробуйте нажать клавишу удаления — DELETE, или воспользоваться комбинацией клавиш CONTROL+H. Если и эти клавиши не работают, введите команду stty[1] для настройки клавиш стирания и удаления строки (см. раздел «Удаление строки» на с. 49):

```
$ stty ek
```

Дополнительная информация по утилите stty приведена в части V.

---

[1] Команда stty является аббревиатурой предложения «set teletypewriter», то есть «настройка пишущего устройства телетайпа» — первого терминала, запускавшегося под управлением UNIX. Сегодня stty обычно применяется для настройки терминала.

Удаление слова

Введенное слово можно удалить, воспользовавшись комбинацией клавиш CONTROL+W. Под словом здесь понимается любая последовательность символов, не содержащая символы пробела или табуляции. Когда нажимается комбинация клавиш CONTROL+W, то при удалении слова курсор перемещается влево к началу текущего слова (если вы находились в процессе ввода слова) или к предыдущему слову (если вы только что ввели символ пробела или табуляции).

CONTROL+Z ПРИОСТАНАВЛИВАЕТ РАБОТУ ПРОГРАММЫ

Хотя это и не относится к исправлению ошибок, вы можете при возникновении ошибки нажать клавишу приостановки (обычно в ее роли выступает комбинация CONTROL+Z) и посмотреть, что при этом произойдет. Если система отображает сообщение, содержащее слово Stopped (остановлено), вы только что остановили выполнение задания, воспользовавшись системой управления заданием. Для продолжения выполнения задания в активном режиме нужно ввести команду fg, и вы вернетесь к тому месту, где находился процесс выполнения задания до нажатия клавиши приостановки. Дополнительные сведения даны в разделе «Перевод задания из активного в фоновый режим выполнения» на сю 155.

Удаление строки

В любой момент до нажатия клавиши ВВОД вы можете удалить набранную строку, нажав клавишу удаления строки. При нажатии этой клавиши курсор перемещается влево, стирая все символы на своем пути, и возвращается к началу строки ввода. По умолчанию в роли клавиши удаления строки выступает комбинация клавиш CONTROL+U. Если эта комбинация не работает, попробуйте комбинацию CONTROL+X. Если и эта комбинация не работает, введите команду stty, рассмотренную в разделе «Удаление символа» на с.48.

Отмена выполнения

Иногда возникают потребности в прекращении работы запущенной программы. К примеру, нужно остановить программу, выполняющую длинную задачу, например отображение содержимого файла длиной в несколько сотен страниц или копирование не того файла, который нужно было скопировать.

При использовании текстового дисплея для прекращения работы программы нужно нажать клавиши прерывания (CONTROL+C или, иногда, DELETE или DEL). Если в качестве клавиши прерывания захочется использовать какую-нибудь другую клавишу, обратитесь к разделу «Специальные клавиши и характеристики» на с. 841 в части V.

Если таким способом прекратить работу программы не получается, попробуйте остановить программу клавишей приостановки (комбинация CONTROL+Z), введите команду jobs, чтобы проверить номер задания запущенной программы, и используйте команду kill, чтобы отменить задание. Номер задания выводится в квадратных скобках в крайней левой позиции той строки, в которой отображается задание (в следующем примере это [1]). В приводимом ниже примере пользователь использует в команде kill ключ -TERM, чтобы отправить сигнал прекращения заданию с указанным номером, перед которым ставится знак процента (%1). В этой команде ключ -TERM можно опустить, поскольку kill отправляет сигнал на прекращение задания по умолчанию.

```
$ bigjob
CONTROL+Z
[1]+  Stopped                 bigjob
```

```
$ jobs
[1]+  Stopped                 bigjob
$ kill -TERM %1
$ ВВОД
[1]+  Killed                  bigjob
```

Команда kill возвращает приглашение. Нажмите еще раз клавишу ВВОД, чтобы посмотреть на подтверждающее сообщение. Если посланный с терминала сигнал прерывания не сработает, подождите десять секунд и нажмите клавишу ВВОД. Если сообщение, свидетельствующее о прекращении задания, не появится, воспользуйтесь сигналом на прекращение (-KILL). Работающая программа обычно не может проигнорировать сигнал на прекращение, он практически гарантирует прекращение работы программы. Дополнительная информация изложена в разделе «Запуск команды в фоновом режиме» на с. 154 и в описании команды kill на с. 738.

При нажатии клавиши прерывания операционная система Linux отправляет с терминала сигнал прерывания в адрес запущенной вами программы и в адрес оболочки. Какой конкретно будет реакция программы, зависит от нее самой.

Может произойти немедленная остановка и выход из программы (что чаще всего и происходит), программа может проигнорировать сигнал, или она может получить сигнал и произвести другое действие, вызванное специальной процедурой обработки сигнала.

У оболочки Bourne Again Shell имеется отдельная процедура обработки сигнала прерывания, поступающего с терминала. Действия этого обработчика сигнала зависят от того, ожидает ли оболочка от пользователя завершения ввода команды, или она ожидает завершения выполнения программы. Если оболочка ожидает ввода от пользователя, обработчик сигнала сбрасывает частично введенную команду и отображает приглашение. Если оболочка ожидает завершение работы программы, обработчик сигнала не предпринимает никаких действий; оболочка продолжает находиться в режиме ожидания. Когда выполнение программы завершается, оболочка выводит приглашение.

Повторение и редактирование командных строк

Для повторения предыдущей команды, введенной в оболочке bash или в оболочке tcsh, следует нажать клавишу СТРЕЛКА ВВЕРХ. При каждом нажатии этой клавиши оболочка выводит ранее введенную командную строку. Чтобы еще раз выполнить команду, появившуюся в командной строке, следует нажать клавишу ВВОД. Чтобы просмотреть командные строки в обратном порядке, следует нажать клавишу СТРЕЛКА ВНИЗ.

Клавиши СТРЕЛКА ВПРАВО и СТРЕЛКА ВЛЕВО перемещают курсор вперед и назад по отображаемой командной строке. Новый символ можно впечатать в любую позицию командной строки. Для удаления символов из командной строки можно воспользоваться клавишей стирания.

Более сложные приемы редактирования командной строки рассмотрены в описаниях оболочки bash в главе 8 и оболочки tcsh в главе 9.

su/sudo: Ограничение ваших полномочий (root-права)

В системах UNIX и Linux всегда есть привилегированный пользователь по имени **root**. Когда вы работаете с правами пользователя **root** (то есть работаете с *root-правами*),

вам предоставляются особые полномочия, распространяющиеся на всю систему. Пользователя, работающего с root-правами, иногда называют *привилегированным пользователем*, или *администратором*. При работе с root-правами можно читать информацию из любого файла и вести запись в любой файл системы, а также выполнять программы, недоступные обычным пользователям. На многопользовательских системах вам могут запретить получение root-прав, не давая тем самым запускать определенные программы. А вот кому-то, в частности *системному администратору*, такая возможность предоставляется, и этот человек занимается обслуживанием системы.

НЕ ЭКСПЕРИМЕНТИРУЙТЕ ПРИ РАБОТЕ С ROOT-ПРАВАМИ

Если вы не обладаете root-правами, можете смело экспериментировать. А когда вы работаете с root-правами, выполняйте только строго необходимую работу и всегда отдавайте отчет своим действиям.

После завершения текущего задания вернитесь к работе под своим собственным именем. При работе с root-правами вы можете нанести такой вред системе, что для восстановления ее работоспособности может понадобиться переустановка Linux.

При обычной установке вы получаете root-права одним из двух способов. Вы можете войти в систему как пользователь с именем **root**. После этого вы будете работать с root-правами, пока не выйдете из системы. Также можно, работая под своим именем, воспользоваться утилитой su (substitute user — сменить пользователя), выполнить одну команду, обладая root-правами, или временно заполучить root-права, чтобы иметь возможность выполнения нескольких команд. Вход в систему под именем root и запуск утилиты su для получения root-прав потребуют от вас ввода пароля, связанного с именем пользователя root. В следующем примере показано, как можно воспользоваться утилитой su для выполнения одной команды.

```
$ ls -l /lost+found
ls: cannot open directory /lost+found: Permission denied
                    Невозможно открыть каталог /lost+found: нет прав доступа
$ su -c 'ls -l /lost+found'
Password:                          Введите пароль доступа для пользователя root
total 0
$
```

Первая команда в предыдущем примере показала, что пользователю, не обладающему root-правами, запрещено выводить список файлов каталога /lost+found: команда ls отображает сообщение об ошибке. Во второй команде используется утилита su с ключом -c (command — команда) для выполнения той же команды с root-правами. Команда взята в одинарные кавычки, чтобы оболочка смогла ее правильно интерпретировать. Когда завершается выполнение команды (ls показывает, что в этом каталоге нет файлов), у пользователя больше нет root-прав. При вызове без аргументов утилита su порождает вызов нового экземпляра оболочки для работа с root-правами. Обычно при работе с root-правами оболочка отображает приглашение в виде знака решетки (#). Для возвращения к обычному знаку приглашения и к обычным правам следует ввести команду exit.

```
$ su
Password:                          Введите пароль доступа для пользователя root
# ls -l /lost+found
```

```
total 0
# exit
exit
$
```

Некоторые дистрибутивы (например, Ubuntu) поставляются с заблокированной учетной записью root (в них отсутствует пароль для пользователя по имени root) и рассчитывают на использование утилиты sudo (www.sudo.ws), чтобы дать возможность пользователям приобрести root-права. Утилита sudo требует от вас для получения root-прав ввести свой пароль (а не пароль root). В следующем примере показано, как пользователь приобретает root-права для просмотра содержимого каталога /lost+found.

```
$ sudo ls -l /lost+found
[sudo] password for sam:          Введите ваш пароль
total 0
$
```

При использовании ключа -i утилита sudo порождает вызов нового экземпляра оболочки для работы с root-правами. Обычно при работе с root-правами оболочка отображает приглашение в виде знака решетки (#). Для возвращения к обычному знаку приглашения и к обычным правам следует ввести команду exit.

```
$ sudo -i
[sudo] password for sam:          Введите ваш пароль
# ls -l /lost+found
total 0
# exit
logout
$
```

Где найти документацию

Дистрибутивы Linux обычно не поставляются с печатными справочными руководствами. Но размещение в Интернете полной документации по системе всегда было сильной стороной Linux. С самых ранних выпусков операционной системы появилась возможность получить доступ к страницам man (manual — руководство) и info, воспользовавшись утилитами man и info. Не удивительно, что при таком бурном росте популярности и системы Linux, и Интернета количество источников документации также неуклонно росло. В этом разделе рассматриваются некоторые места, где можно найти информацию по операционной системе Linux.

Ключ – –help

Многие GNU-утилиты имеют ключ --help, при использовании которого отображается информация об утилите. В утилитах, не имеющих отношения к GNU, для отображения справочной информации могут использоваться ключи -h или -help.

```
$ cat --help
Usage: cat [OPTION] [FILE]…
Concatenate FILE(s), or standard input, to standard output.
  -A, --show-all           equivalent to -vET
  -b, --number-nonblank    number nonblank output lines
```

```
  -e                        equivalent to -vE
  -E, --show-ends           display $ at end of each line
...
```

Информация, отображаемая с помощью ключа `--help`, выходит за пределы экрана, поэтому выходную информацию нужно пропускать через утилиту разбиения на страницы less, используя для этих целей канал:

```
$ ls --help | less
```

man: Отображение системного руководства

Утилита man (см. часть V) отображает man-страницы из системной документации в текстовой среде. Пользу от этой документации можно получить, если знать, какую утилиту вы собираетесь использовать, но не помните в точности порядок ее применения. К man-страницам можно также обратиться за получением информации по определенным темам или для определения функциональных возможностей, доступных в Linux. Поиск тем, имеющихся на man-страницах, можно вести с помощью утилиты apropos. Поскольку описание в системной документации дается в сжатом виде, эта информация будет особенно полезной, если вы уже разбираетесь в основном предназначении той или иной утилиты.

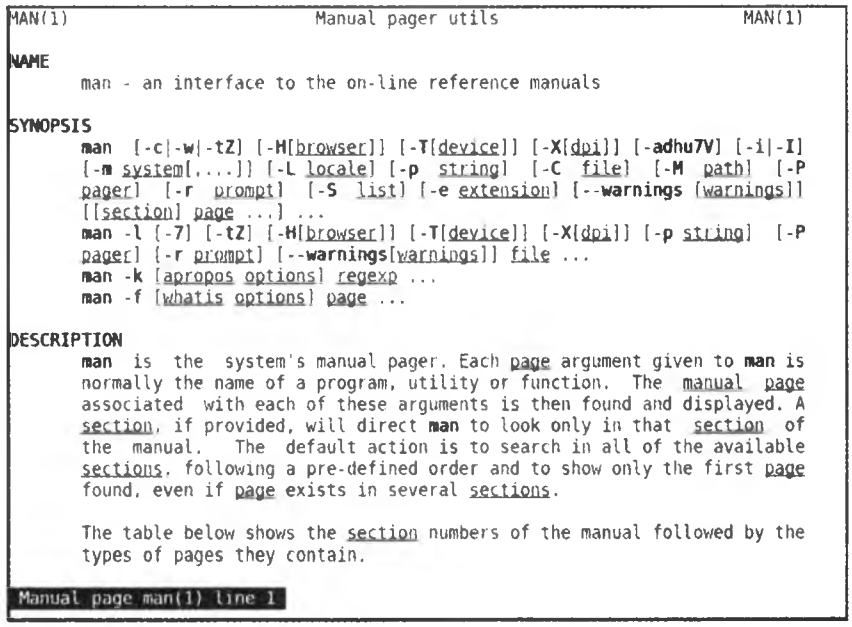

Рис. 2.1. Утилита man, отображающая информацию о самой себе

ИНТЕРНЕТ-СТРАНИЦЫ MAN

На сайте http://tldp.org/manpages/man.html содержатся ссылки на копии большей части man-страниц. Кроме того, страницы представлены в простом HTML-формате, для которого не требуется устанавливать специальную утилиту для чтения man-страниц.

Чтобы найти дополнительные сведения об утилите, нужно ввести команду man, за которой следует имя утилиты. На рис. 2.1 показана команда man, отображающая информацию о себе самой; пользователь ввел команду `man man`.

Команда постраничного вывода less

Утилита man автоматически пропускает свою выходную информацию через команду постраничного вывода, обычно в этой роли выступает команда less, осуществляющая поэкранное отображение информации. При таком доступе к странице руководства less отображает внизу экрана, после того как выведет весь экран текста, приглашение, например `Manual page man(1) line 1`, и ждет, пока вы не запросите следующий экран текста, нажав клавишу ПРОБЕЛ. Если нажать h (help — помощь), то будет отображен список ключей less. Если нажать q (quit — прекратить), команда останавливает свою работу и заставляет оболочку отобразить приглашение. Дополнительные сведения даны в разделе «Основные утилиты» на с. 66.

Разделы руководства

Основываясь на стандарте FHS (Filesystem Hierarchy Standard — стандарт иерархии файловой системы), руководство по системе Linux и man-страницы разбиты на десять разделов, и в каждом разделе описываются соответствующие инструментальные средства:

1. User Commands (Команды пользователя).
2. System Calls (Системные вызовы).
3. Subroutines (Подпрограммы).
4. Devices (Устройства).
5. File Formats (Форматы файлов).
6. Games (Игры).
7. Miscellaneous (Разное).
8. System Administration (Администрирование системы).
9. Kernel (Ядро).
10. New (Новое).

Эта структура очень напоминает структуру набора руководств по UNIX. Если не определить раздел руководства, man отобразит самое первое встреченное в руководстве слово, указанное в командной строке. Многие пользователи находят нужную им информацию в разделах 1, 6 и 7; программисты и системные администраторы обычно нуждаются в информации других разделов.

Иногда в руководстве встречаются статьи для разных инструментальных средств с совпадающими именами. Например, следующая команда отображает man-страницу для утилиты passwd из раздела 1 системного руководства:

```
$ man passwd
```

Чтобы просмотреть man-страницу для файла passwd из раздела 5, нужно ввести следующую команду:

```
$ man 5 passwd
```

Предыдущая команда приказывает man просмотреть на наличие man-страниц только раздел 5. В документации можно увидеть, что ссылка на эту man-страницу имеет вид passwd(5). Чтобы просмотреть все man-страницы по указанной теме, нужно воспользоваться ключом `-a` (adjacent — связанное), чтобы отобразилась следующая страница руководства, нужно нажать клавиши q и ВВОД. Например, для просмотра всех страниц руководства, имеющих отношение к слову `passwd`, нужно ввести команду `man -a passwd`.

КЛЮЧИ

Ключи изменяют способ работы утилиты или команды. Обычно ключи указываются в виде одной или нескольких букв, перед которыми ставится один или два дефиса. Ключи, как правило, ставятся после имени вызываемой утилиты и пробела. Остальные аргументы команды следуют за ключом и пробелом. Дополнительные сведения даны в разделе «Ключи» на с. 138.

apropos: Поиск по ключевому слову

Когда неизвестно имя команды, необходимой для выполнения конкретной задачи, то для ее поиска можно воспользоваться утилитой apropos с необходимым для поиска ключевым словом. Эта утилита ищет ключевое слово в краткой строке описания (которая находится вверху страницы) на всех man-страницах и отображает те из них, которые содержат соответствующее слово. При вызове утилиты man с ключом -k (keyword — ключевое слово) она выдает такую же информацию, что и утилита apropos.

База данных, используемая утилитой apropos, которая называется whatis, недоступна во многих Linux-системах при их первой установке, но она создается автоматически с помощью утилиты cron (сведения о cron даны в части V в разделе, посвященном crontab).

В следующем примере показана информация, выводимая утилитой apropos при вызове ее с ключевым словом who. Эта информация включает имя каждой команды, номер раздела руководства, в котором она содержится, и краткое описание из заголовка man-страницы. Этот список включает необходимую вам утилиту (who) и показывает другие связанные с этим ключевым словом инструментальные средства, которые могут быть вам полезны:

```
$ apropos who
at.allow (5)         - determine who can submit jobs via at or batch
at.deny (5)          - determine who can submit jobs via at or batch
from (1)             - print names of those who have sent mail
w (1)                - Show who is logged on and what they are doing.
w.procps (1)         - Show who is logged on and what they are doing.
who (1)              - show who is logged on
whoami (1)           - print effective userid
```

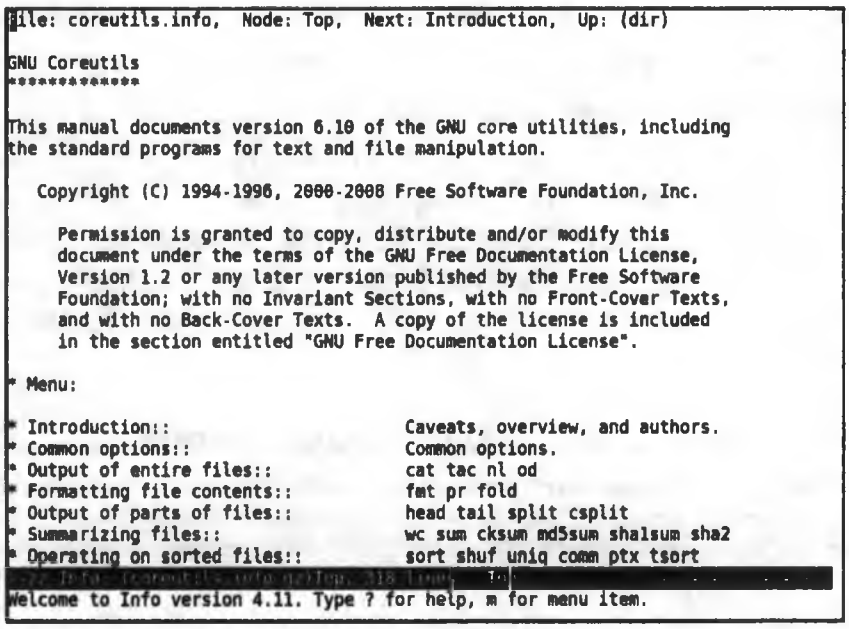

Рис. 2.2. Исходный экран, отображающий информацию по команде info coreutils

whatis

Утилита whatis похожа на apropos, но находит только целое слово, соответствующее имени утилиты:

```
$ whatis who
who                  (1)  - show who is logged on
```

info: Отображение информации об утилитах

Утилита info, работающая в текстовом режиме, является гипертекстовой системой, основанной на использовании меню, разработанной в рамках проекта GNU и распространяемой вместе с Linux. Утилита info может отобразить документацию по многим оболочкам Linux, утилитам и программам, разработанным по проекту GNU. Руководство по info выложено по адресу www.gnu.org/software/texinfo/manual/info. На рис. 2.2 показан экран, отображаемый утилитой info после ввода команды `info coreutils` (coreutils — это пакет программ, в котором содержатся основные утилиты Linux).

УТИЛИТЫ MAN И INFO ОТОБРАЖАЮТ РАЗНУЮ ИНФОРМАЦИЮ

Утилита info отображает более полную и более свежую информацию о GNU-утилитах по сравнению с той информацией, которая отображается утилитой man. Когда man-страница отображает краткую информацию об утилите, о которой есть сведения в info, man-страница ссылается на info. Утилита man часто отображает лишь ту информацию, которая доступна по утилитам, не относящимся к проекту GNU. Когда info отображает информацию по утилитам, не имеющим отношения к GNU, она зачастую представляет собой копию man-страницы.

Поскольку информация на экранах взята из редактируемых файлов, изображение на ваших дисплеях может отличаться от того, которое показано в этом разделе. При отображении исходного info-экрана можно нажать любую из следующих клавиш или клавишных комбинаций:

- ? — выводит список команд info;
- ПРОБЕЛ — вызывает прокрутку меню тех элементов, о которых имеется информация;
- m — если после нажатия этой клавиши ввести имя меню, выводит меню с этим именем;
- m — если сразу после нее нажать ПРОБЕЛ, выводит список меню;
- q или CONTROL+C прекращает работу info.

Система обозначений info, используемая для описания клавиш клавиатуры, может быть вам незнакома. Обозначение C-h — это то же самое, что CONTROL+H. По аналогии с этим, M-x означает удерживание клавиши META или клавиши ALT и нажатие клавиши x. На некоторых системах, чтобы продублировать функцию META+X, нужно нажать ESCAPE, а затем x. Дополнительные сведения даны в разделе «Клавиши: система записи обозначений и использование» на с. 231.

ВОЗМОЖНО, ВМЕСТО INFO БУДЕТ ПРОЩЕ ИСПОЛЬЗОВАТЬ PINFO

Утилита pinfo похожа на info, но она интуитивно понятнее для тех, кто не знаком с редактором emacs. Эта утилита, как и info, работает в текстовом режиме. Если это доступно, то pinfo для облегчения использования своего интерфейса применяет цветовое оформление. Перед использованием утилиты pinfo ее нужно установить.

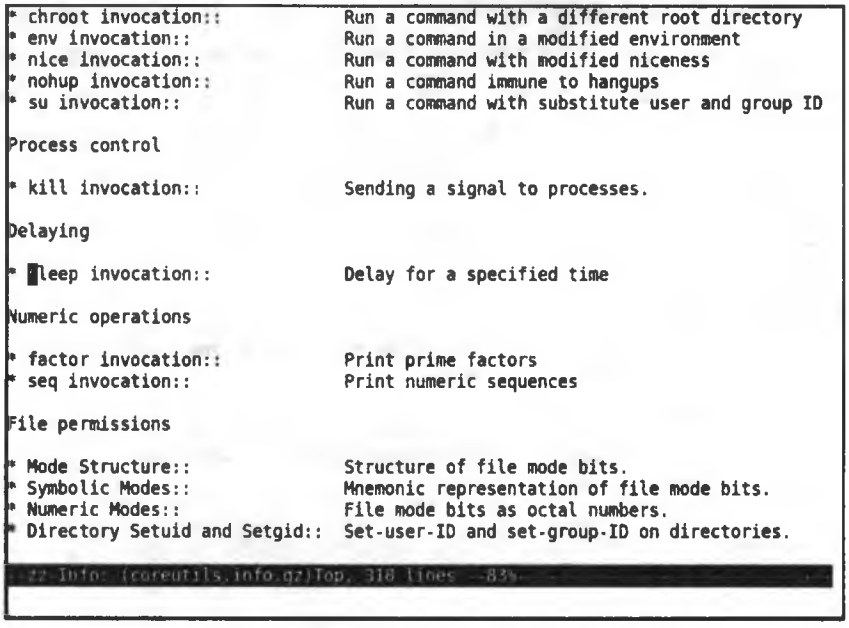

Рис. 2.3. Экран info coreutils, отображаемый после двойного ввода /sleepВВОД

После ввода команды `info coreutils` наберите `/sleep`**ВВОД**, чтобы найти строку `sleep`. После ввода символа / курсор перемещается в самую нижнюю строку экрана, где отображается сообщение о поиске `Search for string [`*строка*`]`: где в качестве *строки* выводится последняя строка, которую вы искали. Для поиска этой же строки следует нажать клавишу **ВВОД** или набрать строку, которую нужно найти. При вводе слова `sleep` оно появляется в этой строке, а после нажатия клавиши **ВВОД** отображается следующий встреченный фрагмент текста со словом `sleep`.

Затем для поиска в тексте следующего слова `sleep` (рис. 2.3) нужно набрать комбинацию /**ВВОД** (или `/sleep`**ВВОД**). Звездочка в крайней левой позиции строки свидетельствует о том, что эта запись является пунктом меню. После звездочки следует имя пункта меню, два двоеточия и описание этого пункта.

Каждый пункт меню — это ссылка на ту info-страницу, на которой дается описание соответствующего элемента. Для перехода на эту страницу нужно воспользоваться **СТРЕЛКАМИ**, чтобы переместить курсор на строку, содержащую пункт меню, и нажать клавишу **ВВОД**. Вместо этого можно набрать название пункта меню в команде menu и просмотреть нужную информацию. Например, чтобы отобразить информацию по утилите sleep, можно ввести команду `m sleep`, нажав после этого клавишу **ВВОД**. При вводе символа m (означающего menu), курсор перемещается к последней строке окна (точно так же, как и при нажатии клавиши /) и отображает в этой строке приглашение `Menu item:`. При вводе слова `sleep` оно отображается в этой строке, а после нажатия клавиши **ВВОД** отображается информация о выбранном вами пункте меню.

На рис. 2.4 показан верхний узел информации по утилите sleep. Группы узла — это набор информации, который может быть прокручен с помощью клавиши **ПРОБЕЛ**. Для отображения следующего узла необходимо нажать клавишу n, а для отображения предыдущего узла — клавишу p. (У пункта sleep есть только один узел.) При чтении данной книги и изучении новых утилит вы можете использовать утилиты man или info для поиска дополнительных сведений об этих утилитах. Если имеется возмож-

ность распечатки PostScript-документов, вы можете вывести на печать страницу руководства, воспользовавшись утилитой man с ключом -t. Например, команда man -t cat | lpr выведет на печать информацию об утилите cat. Для отображения документации, опубликованной на веб-сайте www.tldp.org, можно также воспользоваться веб-браузером и распечатать нужную информацию прямо из него.

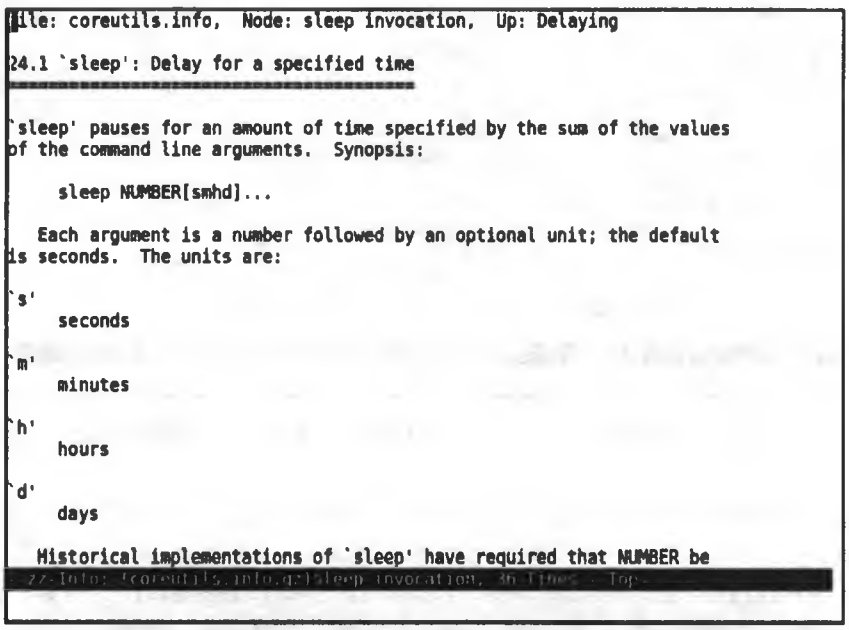

Рис. 2.4. info-страница, посвященная утилите sleep

Документы HOWTO: Информация о том, как все работает

Документ HOWTO дает подробные объяснения того, как можно сделать что-нибудь, имеющее отношение к Linux, — от настройки узлов оборудования и до осуществления задач системного администрирования по настройке сетевых программ. Краткие объяснения изложены в документах Mini-HOWTO. Как и в случае с программным обеспечением Linux, есть человек или группа людей, отвечающая за написание и поддержку документа HOWTO, но вносить в него свой вклад могут многие люди.

Основная часть документов HOWTO и mini-HOWTO размещена на веб-сайте проекта документирования Linux (Linux Documentation Project, LDP). Для доступа к ней нужно войти из веб-браузера на сайт www.tldp.org, щелкнуть на ссылке HOWTOs, и выбрать нужный индекс для поиска HOWTO или mini-HOWTO. Чтобы найти HOWTO и другие документы, можно также воспользоваться поисковой функцией LDP на главной странице сайта.

Получение помощи по работе с системой

В этом разделе приводится описание нескольких мест, где можно найти помощь по использованию системы Linux.

Рис. 2.5. Google выдает сообщение об ошибке

Поиск мест, где находится вспомогательная информация

/usr/share/doc

Каталоги /usr/share/doc и /usr/src/linux/Documentation (которые присутствуют только в случае установки исходного кода ядра) зачастую содержат более подробную и немного другую информацию об утилитах, по сравнению с той информацией, которая предоставляется утилитами man и info. Чаще всего эта информация предназначена для тех, кто будет компилировать и модифицировать утилиту, а не только пользоваться ее возможностями. В этих каталогах содержатся тысячи файлов, в каждом из которых имеется информация по отдельной теме.

Например, следующие команды отображают информацию об утилитах grep и info. Вторая команда, чтобы дать вам возможность просмотра сжатого файла, использует утилиту zcat (см. часть V), при этом файл подвергается распаковке и пропускается через утилиту постраничного просмотра less.

```
$ less /usr/share/doc/grep/README
$ zcat /usr/share/doc/info/README.gz | less
```

Использование для получения помощи ресурсов Интернета

В Интернете можно найти множество полезных сайтов, связанных с Linux. Кроме сайтов в Интернете предлагаются различные виды документации. Можно ввести сообщение об ошибке, полученное из программы, с выполнением которой возникли проблемы, в поисковую систему типа Google (воспользовавшись www.google.com или ее специально настроенной на Linux версией www.google.com/linux). Чтобы повысить качество получаемых результатов поиска, заключите сообщение об ошибке в двойные кавычки. Скорее всего, в результате поиска будут выданы публикации, имеющие отношение к вашей проблеме и содержащие предложения по ее решению (рис. 2.5).

GNU

Руководства GNU доступны по адресу www.gnu.org/manual. Кроме этого можно посетить главную страницу проекта GNU (www.gnu.org) и найти там дополнительную документацию и другие GNU-ресурсы. Многие GNU-страницы и ресурсы доступны на различных языках.

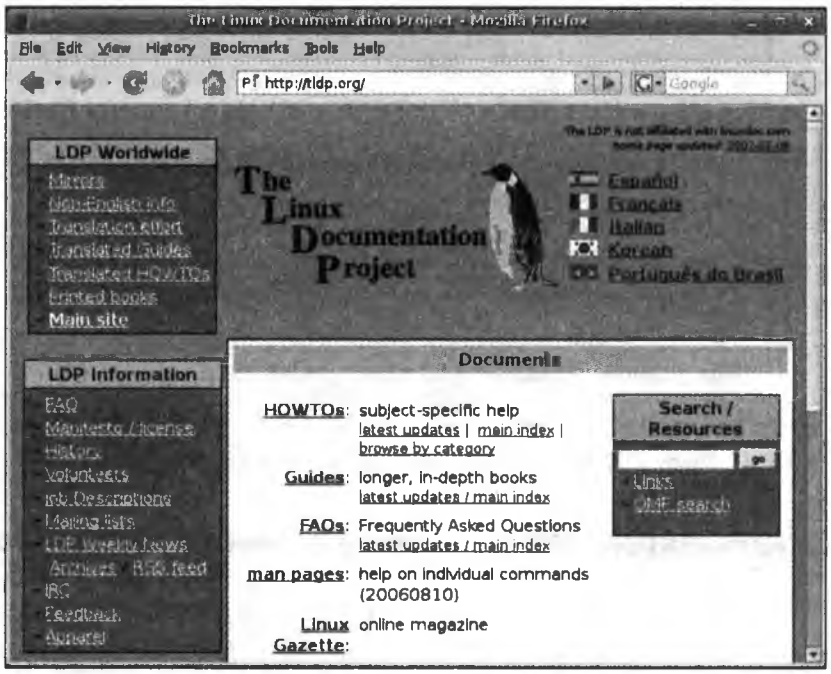

Рис. 2.6. Главная страница Linux Documentation Project

Проект создания документации по Linux

Проект документирования Linux (Linux Documentation Project, www.tldp.org, рис. 2.6), появившийся на свет чуть ли не вместе с самой системой Linux, содержит полный набор руководств, документов HOWTO, ответов на наиболее часто задаваемые вопросы, man-страниц и журналов, посвященных Linux. Главная страница доступна на английском, португальском, испанском, итальянском, корейском и французском языках. Сайт прост в использовании и поддерживает локальный текстовый поиск. Он также предоставляет полный набор ссылок, которые могут использоваться для поиска практически всей, связанной с Linux необходимой вам информации (щелкните на ссылке Links в окне поиска Search или перейдите на страницу www.tldp.org/links). Страница ссылок включает в себя разделы основной информации, описания событий, информации по начальному этапу работы с системой, информации о пользовательских группах, подпискам и новостными группам, и у каждого раздела имеется множество внутренних подразделов.

Дополнительная информация о входе в систему

В этом разделе рассматривается порядок использования виртуальных консолей, последовательность действий при возникновении проблем со входом в систему и способ изменения пароля.

Использование виртуальных консолей

При запуске Linux на персональном компьютере чаще всего приходится работать с дисплеем и клавиатурой, подключенными к этому компьютеру. Используя эту физическую консоль, можно получить доступ сразу к 63 виртуальным консолям (которые также называются виртуальными терминалами). Некоторые из них установлены для входа в систему, а некоторые — для работы в качестве графических дисплеев. Для переключения между виртуальными консолями нужно, удерживая в нажатом состоянии клавиши CONTROL и ALT, нажать функциональную клавишу, связанную с той консолью, информацию которой вы хотите увидеть. Например, CONTROL+ALT+F5 даст возможность отобразить на экране пятую виртуальную консоль. В данной книге та консоль, на которую осуществляется переход по нажатию комбинации клавиш CONTROL+ALT+F1, считается системной консолью или просто консолью.

Обычно активны шесть виртуальных консолей, на которых запущены текстовые сеансы входа в систему. Когда нужно использовать оба интерфейса — текстовый и графический, можно настроить текстовый сеанс на одной из виртуальных консолей, а графический — на другой. Независимо от того, на какой из консолей будет запущен графический сеанс, он будет работать на первой же незадействованной виртуальной консоли (обычно на консоли номер семь).

Что делать, если не удается войти в систему

Если имя пользователя или пароль введены неправильно, то, после того как введены *и* имя пользователя, *и* пароль, система выводит сообщение об ошибке. Это сообщение свидетельствует о том, что вы неправильно ввели либо имя пользователя, либо пароль или что они не соответствую учетной записи. Между неприемлемым именем пользователя и неприемлемым паролем не делается никаких различий — это помогает сбить с толку тех, кто пытается получить несанкционированный доступ к системе путем подбора имен и паролей.

Причин неудачного входа в систему может быть несколько:

- **Имя пользователя и пароль чувствительны к регистру букв.** Убедитесь в том, что режим фиксации верхнего регистра выключен, и введите имя пользователя и пароль именно в том виде, в котором они указаны или в котором вы сами их установили.
- **Попытка войти в систему предпринималась не с той машины.** Комбинация имя пользователя – пароль может быть неверной, если попытка войти в систему предпринимается не с той машины. На более крупной, сетевой системе перед входом в нее может быть нужно указать ту машину, к которой вы хотите подключиться.
- **Введено неверное имя пользователя.** Комбинация имя пользователя – пароль может быть неверной, если вы не зарегистрированы как пользователь. В таком случае следует обратиться к системному администратору.

Выход из системы

При использовании символьного интерфейса для выхода из системы в ответ на приглашение оболочки нужно нажать комбинацию клавиш CONTROL+D. При этом оболочке

посылается сигнал EOF (end of file — конец файла). Кроме этого можно ввести команду exit. Выход из оболочки не приводит к завершению графического сеанса работы; он осуществляется только из той оболочки, с которой вы работали. Например, выход из оболочки, предоставляемой терминалом GNOME, закрывает окно терминала GNOME.

Смена пароля

Если ваш пароль дан вам кем-то другим, то лучше будет установить новый, свой собственный пароль. Из соображений безопасности ни одна из утилит не отображает введенный вами пароль.

ПАРОЛЬ НУЖНО ЗАЩИЩАТЬ

Не давайте никому ни малейшей возможности вскрыть ваш пароль: не записывайте пароль в незашифрованный файл, не позволяйте никому наблюдать за тем, как вы вводите пароль, не передавайте пароль незнакомым людям (системному администратору знать ваш пароль не требуется). Вы всегда можете записать свой пароль и хранить его в безопасном, укромном месте.

ВЫБИРАЙТЕ ПАРОЛЬ, КОТОРЫЙ ТРУДНО УГАДАТЬ

Не используйте телефонных номеров, имен детей и кличек домашних животных, дней рождений, словарных слов (даже на иностранном языке) и т. д. Не пользуйтесь этими же элементами с переставленными символами или формой написания слов, при которой вместо букв используются цифры, воспринимаемые на слух как сочетания букв: современные взломщики, использующие специальные словари, могут применить при подборе и эти, видоизмененные формы слов.

РАЗЛИЧАЙТЕ ВАЖНЫЕ И МЕНЕЕ ВАЖНЫЕ ПАРОЛИ

Возьмите за правило различать важные и менее важные пароли. Например, пароли веб-сайтов для блогов или для доступа к загрузке информации не считаются важными; для подобных сайтов вполне можно обойтись одним и тем же паролем. Но при входе в систему, на почтовый сервер и на свою веб-страницу в банковской системе пароль имеет очень важное значение: для каждой учетной записи из этой категории нужно применять разные пароли, никогда не пользуясь теми же паролями на второстепенных веб-сайтах.

Чтобы изменить пароль, нужно ввести команду passwd. Сначала passwd запросит ваш текущий (старый) пароль. Этот пароль проверяется, чтобы убедиться в том, что изменить ваш пароль не пытается кто-нибудь другой, не имеющий на это полномочий. Затем система запрашивает новый пароль.

ПОДОБРАТЬ НАДЕЖНЫЙ ПАРОЛЬ ПОМОЖЕТ PWGEN

Утилита pwgen (которую, возможно, придется устанавливать) создает список практически случайных паролей. Подключив свое воображение, вы сможете произнести, а следовательно, и запомнить некоторые из этих паролей.

После ввода нового пароля система попросит ввести его еще раз, чтобы убедиться в том, что не допущена ошибка при первом наборе пароля. Если новый пароль оба раза был введен одинаково, ваш пароль будет изменен. Если введенные пароли отличаются друг от друга, это означает, что вы ошиблись при вводе одного из них, и система выведет следующее сообщение об ошибке:

```
Sorry, passwords do not match  К сожалению, пароли не совпадают
```

Если ваш пароль имеет недостаточную длину, система выведет следующее сообщение:

```
You must choose a longer password  Вам следует выбрать более длинный пароль
```

Если пароль слишком прост, система выведет следующее сообщение:

```
Bad: new password is too simple  Неудачный выбор: пароль слишком прост
```

Если система выводит сообщение об ошибке, приходится начинать все сначала. Нажмите несколько раз **ВВОД**, пока оболочка не выведет приглашение, и снова запустите утилиту **passwd**.

После успешной смены пароля меняется порядок вашего входа в систему. Если вы забыли свой пароль, то кто-нибудь, работающий с root-правами, может запустить утилиту **passwd**, чтобы сменить пароль и сообщить вам его новую версию.

При работе под управлением OS X утилита **passwd** изменяет ваш пароль для входа в систему, но не изменяет пароль Keychain. Пароль Keychain используется для различных графических приложений. Изменить его можно, воспользовавшись приложением **Keychain Access**.

Надежные пароли

Относительная надежность паролей обеспечивается наличием в них комбинации цифр, букв в верхнем и нижнем регистре, а также знаков пунктуации. Надежный пароль должен отвечать следующим требованиям:

- В нем должно быть по крайней мере от четырех до шести и более символов (в зависимости от настроек системы администратором). Хорошим компромиссом между длиной и надежностью могут стать семь-восемь символов.
- Пароль не должен быть словом из словаря любого языка, независимо от того, насколько оно кажется вам непонятным.
- В качестве пароля не должны использоваться имя человека, название местности, кличка животного или другие легко угадываемые слова.
- Пароль должен содержать хотя бы две буквы и одну цифру или знак пунктуации.
- Пароль не должен быть вашим именем пользователя, в том числе и в обратном написании или в варианте со смещенными на одну или несколько позиций буквами.

Обязательным здесь является только первое правило. Не используйте управляющие символы (например, **CONTROL+H**), поскольку они могут иметь специальное значение для системы и не позволят вам войти в нее. При смене пароля новый пароль должен отличаться от старого по крайней мере тремя символами. Смена регистра символа не может считаться использованием другого символа.

Заключение

Доступ к системе Linux, как доступ ко многим другим операционным системам, требует авторизации при входе в систему. В ответ на приглашение `login:` требуется ввести свое имя пользователя, а затем ввести пароль. Для смены пароля в любой момент после входа в систему можно воспользоваться утилитой **passwd**. Следует выбирать такой пароль, о содержимом которого трудно догадаться, и который отвечает требованиям, установленным системным администратором.

Системный администратор отвечает за обслуживание системы. На однопользовательских системах его роль достается самому пользователю. На небольших многопользовательских системах за системного администратора будете работать вы или какой-нибудь другой пользователь, или же эта работа будет вестись совместно. На больших многопользовательских или сетевых системах часто работает профессиональ-

ный системный администратор. Когда для решения конкретных системных задач требуются особые права, системный администратор приобретает root-права, войдя в систему под именем root или запустив утилиту su или sudo. На многопользовательских системах root-права могут предоставляться нескольким доверенным пользователям.

Не работайте с root-правами, не имея на то особых причин. Когда нужно сделать что-нибудь, для чего требуются root-права, работайте с этими правами строго необходимое для этого время. Возвращайтесь к работе под своим собственным именем сразу же, как только это станет возможным.

Утилита man дает возможность просматривать интерактивную документацию по системным утилитам. Эта утилита будет полезна как для начинающих, так и для опытных пользователей системы Linux, которые должны глубже изучать системную документацию, черпая из нее сведения о характерных особенностях поведения той или иной утилиты. Утилита info также помогает не только новичкам, но и знатокам и открывает доступ к документации по многим утилитам Linux.

Упражнения

1. При попытке входа в систему под неверным именем пользователя или паролем, выводится следующее сообщение:

   ```
   Login incorrect  При входе в систему допущена ошибка
   ```

   Из этого сообщения непонятно, что именно было неверным, имя пользователя, пароль, или и то и другое. Почему до вас не доводится эта информация?
2. Приведите три примера неудачно выбранных паролей. Что позволяет забраковать каждый из них? Включите в их число слишком короткий пароль. Приведите пример сообщения об ошибке, которое отображается при этом системой.
3. Можно ли fido считать приемлемым паролем? Приведите несколько причин за и против.
4. Что нужно делать, если войти в систему не удается?
5. Попробуйте сменить свой пароль на dog. Что при этом произойдет? А теперь смените его на более надежный пароль. Что обеспечивает паролю относительную надежность?

Упражнения повышенной сложности

1. Измените свой вход в оболочку tcsh, не пользуясь root-правами.
2. Сколько man-страниц в подразделе «Devices» (Устройства) системного руководства?

   (Подсказка: «Devices» является подразделом раздела специальных файлов — «Special Files».)
3. Пример из раздела «apropos: Поиск по ключевому слову» показывает, что man-страницы для слова passwd имеются в разделах 1 и 5 системного руководства. Объясните, как воспользоваться утилитой man, чтобы узнать, в каком разделе системного руководства содержится страница с заданным именем.
4. Как определить, какие утилиты Linux создают архивные файл и работают с ними?

Глава 3. Утилиты

После того как Линус Торвальдс представил Linux, у этой системы еще довольно долгое время не было графического пользовательского интерфейса (GUI): система запускалась с использованием текстового интерфейса, который также назывался интерфейсом командной строки (command-line interface, CLI). Все инструментальные средства запускались только из командной строки. В наши дни графический интерфейс имеет для Linux очень важное значение, но тем не менее многие пользователи, особенно системные администраторы, активно пользуются обширным арсеналом утилит командной строки. Эти утилиты зачастую работают быстрее, обладают большей мощностью и совершенством по сравнению со своими GUI-аналогами. Иногда в GUI просто нет аналогов тех утилит, которые работают в текстовом режиме, а некоторым людям просто нравится иметь в своем распоряжении командную строку.

При работе с интерфейсом командной строки вы работаете с оболочкой (см. главы 5, 8 и 10). Перед началом работы с оболочкой важно разобраться с символами, имеющими для нее специальное значение, поэтому данная глава начинается с рассмотрения специальных символов. Затем в главе дается описание пяти основных утилит: ls, cat, rm, less и hostname. После этого рассматривается ряд других утилит, предназначенных для работы с файлами, а также утилиты, отображающие пользователей, зарегистрировавшихся в системе, утилиты, позволяющие обмениваться данными с другими пользователями, распечатывать, сжимать файлы и возвращать их в обычное состояние, а также запаковывать файлы в архив и извлекать их оттуда.

Специальные символы

К специальным относятся символы, имеющие особое значение для оболочки. Они рассматриваются в разделе «Генерация имени файла и расширение имени» на с. 157. Эти символы упомянуты сейчас, чтобы предостеречь вас от случайного использования их в качестве обычных символов до тех пор, пока вы не поймете, как они интерпретируются оболочкой. Например, лучше избегать использования любого из следующих символов в именах файлов (даже если их использует программа emacs и ряд других программ), поскольку они затрудняют ссылку на файл в командной строке:

```
& ; | * ? ' " ' [ ] ( ) $ < > { } # / \ ! ~
```

Пробельные символы

Хотя символы, генерируемые клавишами ВВОД, ПРОБЕЛ и TAB, и не относятся к специальным, но они имеют для оболочки особое значение. Символом, который генерируется клавишей ВВОД, обычно завершается командная строка и инициируется выполнение команды. Символами пробела и табуляции отделяются друг от друга элементы командной строки, и эти символы обобщенно называются пробельными или пустыми символами.

Цитирование специальных символов

Если символ, имеющий для оболочки специальное значение, нужно использовать в качестве обычного символа, его нужно процитировать (или дезактивировать). При цитировании специального символа обо-

лочка не придает ему специального значения. Она рассматривает его в качестве вполне обычного символа. Но символ слэша (/) всегда используется в качестве разделителя в путевом имени, даже когда вы его цитируете.

Обратный слэш Чтобы процитировать символ, перед ним нужно поставить символ обратного слэша (\). Когда встречаются сразу два и более специальных символов, обратный слэш нужно ставить перед каждым из них (например, символы ** нужно вводить как **). Обратный слэш также можно процитировать как и любой другой специальный символ, поставив перед ним еще один обратный слэш (\\).

Одинарные кавычки Другой способ цитирования специальных символов предусматривает заключение этих символов в одинарные кавычки: '**'. Взятием в одинарные кавычки можно процитировать множество как специальных, так и обычных символов: 'Это специальный символ: >'. При этом обычные символы интерпретируются как и прежде, а специальные символы оболочка интерпретирует как обычные.

Единственный способ цитирования символа стирания (CONTROL+H), символа удаления строки (CONTROL+U) и других управляющих символов (попробуйте CONTROL+M) заключается в установке перед каждым из них символа, генерируемого комбинацией CONTROL+V. Одинарные кавычки и символы обратного слэша в данном случае не работают. Попробуйте ввести следующие символы:

```
$ echo 'xxxxxxCONTROL+U'
$ echo xxxxxxCONTROL+V CONTROL+U
```

Хотя заметить символ, генерируемый комбинацией CONTROL+U, за то мгновение, на которое он появляется при вводе предыдущих двух команд, довольно сложно, он все же присутствует в строке. Следующая команда отправляет выходные данные команды echo через канал команде od (octal display — отображение в восьмеричном виде), чтобы показать символ, генерируемый комбинацией CONTROL+U в виде восьмеричного числа 25 (025):

```
$ echo xxxxxxCONTROL+V CONTROL+U | od –c
0000000 x x x x x x 025 \n
0000010
```

Сочетание символов \n отображает символ новой строки, который команда echo отправляет в конце своей выходной информации.

Основные утилиты

Одно из главных преимуществ системы Linux состоит в том, что она поставляется вместе с тысячами утилит, выполняющих огромное количество функций. В каком бы виде они не применялись, непосредственно из командной строки, или опосредованно, через меню или значки, без них при работе просто не обойтись. В следующем разделе рассматриваются некоторые из наиболее основных и важнейших утилит, доступных при работе с текстовым интерфейсом. Некоторые из них доступны и при работе с графическим интерфейсом, а некоторые другие – только из графического интерфейса.

ЗАПУСКАЙТЕ УТИЛИТЫ ИЗ КОМАНДНОЙ СТРОКИ

В этой главе рассматриваются утилиты, работающие в текстовом режиме, или утилиты командной строки. Экспериментировать с ними можно из терминала, эмулятора терминала, работающего в GUI, или из виртуальной консоли.

Папка или каталог

В следующих разделах широко используется термин *каталог*, обозначающий ресурс, который может содержать файлы. В других операционных системах, включая Windows и Mac OS X, а также довольно часто при упоминании Linux GUI, каталог называют *папкой*. Это вполне удачная аналогия: в традиционной картонной папке, как и в каталоге, хранятся файлы (бумажные дела).

В ЭТОЙ ГЛАВЕ ВЫ РАБОТАЕТЕ В СВОЕМ ЛИЧНОМ КАТАЛОГЕ

При входе в систему вы работаете в своем личном каталоге. В данной главе это единственный используемый каталог: все файлы, создаваемые при ее изучении, будут находиться в вашем личном каталоге. Более подробное описание каталогов дается в главе 4.

ls: Вывод списка имен файлов

Используя избранный вами редактор, создайте небольшой файл по имени practice. (Руководство по редактору vim содержится в главе 6, а руководство по редактору emacs — в главе 7.) Чтобы вывести список имен файлов, имеющихся в вашем личном каталоге, после выхода из редактора можно воспользоваться утилитой ls (list — вывод списка). В первой команде, показанной в листинге 3.1, ls выводит имя файла practice. (Могут также быть выведены имена файлов, созданных системой или программой автоматически.) Следующие команды, показанные в листинге 3.1, отображают содержимое файла и удаляют файл. Эти команды рассматриваются ниже. Дополнительные сведения об этой утилите даются в части V или выводятся по команде `info coreutils 'ls invocation'`.

Листинг 3.1. Использование утилит ls, cat и rm с файлом по имени practice

```
$ ls
practice
$ cat practice
Это небольшой файл, который
я создал в текстовом редакторе
$ rm practice
$ cat practice
cat: practice: No such file or directiory Нет такого файла или каталога
```

cat: Отображение текстового файла

Утилита cat отображает содержимое текстового файла. Имя этой команды происходит от слова *catenate*, означающего последовательное объединение. (В главе 5, в листинге 5.4 показано использование утилиты cat для последовательного объединения содержимого трех файлов.)

Ввод команды cat, пробела и имени файла является удобным способом вывода содержимого файла на экран. В листинге 3.1 показана команда cat, отображающая содержимое файла practice. В этом листинге показана разница между утилитами ls и cat: утилита ls отображает имя файла, а утилита cat отображает содержимое файла. Дополнительные сведения об этой утилите можно найди в части V или введя команду `info coreutils 'cat invocation'`.

rm: Удаление файла

Утилита rm (remove — удалить) удаляет файл. В листинге 3.1 показана утилита rm, удаляющая файл по имени practice. После того как rm удаляет файл, утилиты ls и cat демонстрируют, что файла practice больше в каталоге нет. (Утилита ls не отображает его имени, а утилита cat сообщает о том, что такой файл не существует.) Утилитой rm следует пользоваться крайне осмотрительно. Дополнительные сведения о ней можно найди в части V или введя команду `info coreutils 'rm invocation'`.

БЕЗОПАСНЫЙ СПОСОБ УДАЛЕНИЯ ФАЙЛОВ

Чтобы убедиться в том, что удаляется только тот файл (или файлы), который вы действительно намерены удалить, используйте интерактивную форму вызова утилиты rm. Если после команды rm поставить ключ –i и имя удаляемого файла, rm отобразит имя файла, а затем, перед тем как удалить этот файл, будет ждать вашего ответа y (yes — да). Если вы ответите строкой, которая начинается не с символа y, файл удален не будет.

```
$ rm -i toollist
rm: remove regular file 'toollist'? y
```

Дополнительно. Для команды rm –i можно создать псевдоним (см. главу 8, раздел «Псевдонимы») и поместить его в ваш файл запуска (см. раздел «Файлы запуска» на с.103), чтобы команда rm всегда запускалась в интерактивном режиме.

less и more: Обеспечение поэкранного вывода текстового файла

Утилиты разбиения на страницы

Когда нужно просмотреть файл, не умещающийся на одном экране, можно воспользоваться как утилитой less, так и утилитой more. Каждая из этих утилит делает паузу после отображения экрана текста, а для отображения следующего экрана следует нажать клавишу ПРОБЕЛ. Поскольку эти утилиты обеспечивают постраничную демонстрацию, их называют утилитами разбиения на страницы. При всей своей схожести утилиты less и more все же немного отличаются друг от друга. Например, в конце файла утилита less отображает сообщение `END` и, перед тем как вернуть управление оболочке, ждет, пока вы не нажмете клавишу q. При использовании обеих утилит вы можете нажать клавишу h для отображения экрана помощи со списком команд, которые могут использоваться при разбиении файла на страницы. Введите команды `less practice` и `more practice` вместо команды cat, показанной в листинге 3.1, и посмотрите, как эти команды работают. Если хотите поэкспериментировать с более длинным файлом, воспользуйтесь командой `less /usr/share/dict/words`. Дополнительные сведения об утилитах less и more можно найти в части V или на посвященных им man-страницах.

hostname: Отображение имени системы

Утилита hostname отображает имя системы, на которой вы работаете. Ее следует использовать в том случае, если вы не уверены, что вошли в систему на нужной машине. Дополнительные сведения об этой утилите можно получить на соответствующей man-странице.

```
$ hostname
dog
```

Работа с файлами

В этом разделе рассматриваются утилиты, осуществляющие копирование, перемещение, распечатку, отображение, сортировку и сравнение файлов, а также поиск в них.

АВТОЗАВЕРШЕНИЕ ИМЕНИ ФАЙЛА

После ввода одной и более букв имени файла (которое следует за командой) в командную строку нажмите клавишу TAB, и оболочка Bourne Again Shell, насколько это возможно, дополнит имя файла, завершив его ввод. Если с введенного символа начинается имя только одного файла, оболочка завершит ввод имени файла и поставит за ним символ пробела. Вы можете продолжить ввод или можете нажать клавишу ВВОД для выполнения команды, введенной на данный момент. Если уже введенные символы не идентифицируют имя файла однозначным образом, оболочка дополняет имя, исходя из своих возможностей, и ожидает ввода следующих символов. Если нажатие клавиши TAB не изменяет отображаемую информацию, нажмите TAB еще раз (см. разделы, посвященные автозавершению в оболочке Bourne Again Shell в главе 8) или CONTROL+D (см. разделы, посвященные автозавершению в оболочке TC Shell в главе 9), чтобы отобразить список возможных вариантов автозавершения.

cp: Копирование файла

Утилита cp (copy — копирование) (листинг 3.2) создает копию файла. Эта утилита может скопировать любой файл, включая текстовые и двоичные (исполняемые программы). Утилиту cp можно использовать для создания резервной копии файла или копии для экспериментов.

В командной строке, вызывающей утилиту cp, используется следующий синтаксис, в котором указываются имя копируемого файла и файла-копии:

```
cp файл_источник файл_приемник
```

Файл_источник — это имя файла, копируемого с помощью утилиты cp, а *файл_приемник* — это имя, которое утилита cp присваивает создаваемой (новой) копии файла.

Листинг 3.2. Команда cp копирует файл

```
$ ls
memo
$ cp memo memo.copy
$ ls
memo memo.copy
```

Командная строка вызова утилиты cp, показанная в листинге 3.2, приводит к копированию файла memo в файл memo.copy. Точка является частью имени файла, то есть еще одним символом. Первая команда ls показывает, что memo является единствен-

ным файлом каталога. После выполнения команды cp вторая команда ls показывает наличие в каталоге уже двух файлов: memo и memo.copy.

Иногда в имя копии файла удобно встраивать дату. В следующем примере в имя копии файла включается дата 30 января (0130):

```
$ cp memo memo.0130
```

Хотя для Linux это не имеет никакого значения, дата может помочь вам найти версию файла, созданную в определенный день. Включение даты может также помочь избежать перезаписи уже существующих файлов, предоставляя им каждый день уникальное имя. Дополнительные сведения изложены в разделе «Имена файлов» на с. 99.

Дополнительную информацию по утилите cp можно найти в части V и на info-странице, вызываемой по команде `info coreutils 'cp invocation'`. Если нужно скопировать файл из одной системы в другую в обычной сети, следует воспользоваться утилитами scp, ftp или rsync, рассмотренными в части V.

УТИЛИТА CP МОЖЕТ УНИЧТОЖИТЬ ФАЙЛ

Если файл-копия уже существовал до того, как была запущена команда cp, ее запуск приведет к перезаписи этого файла. Поскольку команда cp перезаписывает существующий файл-приемник (и уничтожает его прежнее содержимое) без предупреждения, вы должны проявлять особую осторожность, чтобы cp не уничтожила ценный для вас файл. Команда cp с ключом –i (interactive — работа в интерактивном режиме) выводит перед перезаписью файла запрос-приглашение на подтверждение действия.

В следующем примере предполагается, что файл orange.2 существует до ввода команды cp. Чтобы перезаписать содержимое файла, пользователь в ответ нажимает клавишу «y»:

```
$ cp -i orange orange.2
cp: overwrite 'orange.2'? y
```

mv: Изменение имени файла

Утилита mv (move — переместить) переименовывает файл без создания его копии. В командной строке, вызывающей утилиту mv, указывается существующий файл и новое имя файла и используется синтаксис, аналогичный тому, который использовался для команды cp:

```
mv имя_существующего_файла новое_имя_файла
```

Командная строка в листинге 3.3 изменяет имя файла memo на memo.0130. Первая команда ls показывает, что memo является единственным файлом каталога. После запуска команды mv в каталоге появляется единственный файл по имени memo.0130. Сравните этот результат с тем, который был показан в листинге 3.2 после запуска команды cp.

Утилита mv может использоваться не только для изменения имени файла. Дополнительные сведения даны в разделе «mv, cp: Перемещение или копирование файлов» на с. 110. Также можно просмотреть раздел части V, посвященный этой команде, или ввести команду `info coreutils 'mv invocation'`.

УТИЛИТА MV МОЖЕТ УНИЧТОЖИТЬ ФАЙЛ

Точно так же, как и cp, утилита mv может уничтожить файл. Так же, как и у cp, у mv есть ключ –i (interactive). См. выше врезку «Утилита cp может уничтожить файл».

Листинг 3.3. Переименование файла с помощью утилиты mv

```
$ ls
memo
$ mv memo memo.0130
$ ls
memo.0130
```

lpr: Распечатка файла

Утилита lpr (line printer — построчный принтер) помещает один или несколько файлов в очередь на распечатку. Системой Linux поддерживается очередь печати, благодаря которой на каждом отдельно взятом принтере одновременно происходит распечатка только одного задания. Очередь позволяет нескольким пользователям или задачам одновременно отправлять выходную информацию на один принтер при вполне ожидаемых результатах. Чтобы вывести список доступных принтеров (на системах, имеющих доступ к более чем одному принтеру), можно воспользоваться командой `lpstat -p`. Чтобы заставить утилиту lpr поместить файл в очередь указанного принтера, даже если он находится на другой (подключенной к сети) системе, нужно воспользоваться ключом `-P`. Следующая команда выводит на печать файл по имени report:

```
$ lpr report
```

Поскольку в этой команде принтер не указан, выходная информация поступает на принтер, используемый по умолчанию. Если к системе подключен только один принтер, то по умолчанию используется именно он.

Следующая строка содержит команду, выводящую тот же файл на печать на принтере по имени `mailroom`:

```
$ lpr -P mailroom report
```

О том, какое задание выводится на печать из очереди печати, можно узнать с помощью ввода команды `lpstat -o` или с помощью утилиты lpq:

```
$ lpq
lp is ready and printing
Rank  Owner   Job Files                 Total Size
active max     86 (standard input)       954061 bytes
```

В данном примере у пользователя Max имеется одно задание, выводимое на печать, никаких других заданий в очереди нет. Чтобы удалить задание из очереди на печать и остановить связанную с ним распечатку, можно воспользоваться номером задания (в данном случае — 86) и утилитой lprm:

```
$ lprm 86
```

Одной командой можно отправлять на принтер более одного файла. В следующей строке приводится команда на распечатку трех файлов на принтере с именем `laser1`:

```
$ lpr -P laser1 05.txt 108.txt 12.txt
```

Дополнительные сведения об утилите lpr можно найти в разделе «lpr» главы 4 или на соответствующей man-странице.

grep: Поиск строки

Утилита grep[1] осуществляет поиск в одном или нескольких файлах указанной строки символов. Эта утилита не вносит изменений в файл, в котором ведется поиск, а просто отображает каждую строку файла, содержащую найденную символьную строку.

Команда grep в листинге 3.4 осуществляет в файле memo поиск тех строк, в которых содержится символьная строка `credit`, и отображает одну строку, соответствующую этому критерию. Если в файле memo содержатся такие слова, как discredit, creditor или accreditation, утилита grep также отобразит и те строки, в которых они содержатся, поскольку они содержат искомую символьную строку. Ключ `-w` (words — слова) заставляет утилиту grep искать соответствия только в виде целых слов. Хотя заключать искомую строку в одинарные кавычки не требуется, но если это сделать, в строке поиска можно будет использовать пробелы и специальные символы.

Листинг 3.4. Использование утилиты grep для поиска строки

```
$ cat memo
Helen:
In our meeting on June 6 we
discussed the issue of credit.
Have you had any further thoughts
about it?
            Max
$ grep 'credit' memo
discussed the issue of credit.
```

Утилита grep обладает более широкими возможностями, чем поиск простой строки в отдельно взятом файле. Дополнительные сведения доступны в части V или на соответствующей man-странице.

head: Отображение начальных строк файла

Утилита head отображает (по умолчанию) первые десять строк файла. Эту утилиту можно использовать, чтобы по выводимым строчкам вспомнить о содержимом конкретного файла. Например, если есть файл по имени months, в котором в календарном порядке перечисляются все 12 месяцев, по одному в каждой строке, то утилита head выведет месяцы с `Jan` по `Oct` (листинг 3.5).

Листинг 3.5. Утилита head выводит первые строки файла, а утилита tail выводит последние строки файла

```
$ head months
Jan
Feb
Mar
Apr
```

[1] Изначально имя grep было игрой слов, используемой в редакторе ed — исходном UNIX-редакторе, доступном в Linux. Эта игра была связана с командой g/re/p. В этой команде g означает global (глобальный режим), re — регулярное выражение, выделенное с помощью слэшей, а p означает вывод (print).

```
May
Jun
Jul
Aug
Sep
Oct
$ tail -5 months
Aug
Sep
Oct
Nov
Dec
```

Утилита head может вывести любое количество строк, поэтому ее можно использовать для просмотра только первой строки файла, а также полного экрана и т. д. Чтобы указать количество выводимых строк, нужно включить в команду дефис, за которым указывается количество выводимых строк. Например, следующая команда выводит только первую строку файла months:

```
$ head -1 months
Jan
```

Утилита head может также вывести фрагменты файла не по заданному количеству строк, а на основе количества блоков или символов. Дополнительные сведения можно найти в части V или введя команду `info coreutils 'head invocation'`.

tail: Отображение данных, находящихся в конце файла

Утилита tail похожа на утилиту head, но в отличие от нее по умолчанию выводит десять последних строк файла. В зависимости от способа ее вызова, эта утилита может вывести меньше или больше, чем десять строк, а также использовать не количество строк, а количество блоков данных или символов для отображения части файла и выводить строки, добавляемые к изменяемому файлу.

Команда tail в листинге 3.5 отображает последние пять строк (с Aug по Dec) файла months.

С помощью следующей команды можно отслеживать строки по мере их добавления к концу дополняемого файла по имени logfile:

```
$ tail -f logfile
```

Чтобы остановить выполнение команды tail и получить приглашение оболочки, следует нажать клавишу прерывания (обычно это комбинация CONTROL+C). Дополнительные сведения можно найти в части V или введя команду `info coreutils 'tail invocation'`.

Листинг 3.6. Утилита sort отображает строки файла в определенном порядке

```
$ cat days
Monday
Tuesday
Wednesday
Thursday
Friday
```

продолжение ⇩

Листинг 3.6 (*продолжение*)

```
Sunday
$ sort days
Friday
Monday
Saturday
Sunday
Thursday
Tuesday
Wednesday
```

sort: Отображение содержимого файла в определенном порядке

Утилита sort выводит содержимое файла построчно в определенном порядке, не изменяя при этом его содержимое. В листинге 3.6 показана команда cat, которая выводит содержимое файла по имени days, состоящего из названий каждого дня недели, занимающих отдельные строки и следующих в календарном порядке. Затем утилита sort отображает содержимое файла в алфавитном порядке.

Утилиту sort можно использовать для выстраивания списков в определенном порядке. Ключ `-u` позволяет генерировать отсортированный список, в котором каждая строка является уникальной (не имеет дубликатов). Ключ `-n` позволяет выстроить список чисел в порядке возрастания. Дополнительные сведения можно найти в части V или введя команду `info coreutils 'sort invocation'`.

uniq: Удаление повторяющихся строк из отображения файла

Утилита uniq (unique — уникальность) выводит файл, пропуская соседние дублированные строки, но при этом не изменяя исходного файла. Если файл содержит список имен, имеющий две последовательные записи для одного и того же человека, uniq пропустит строку-дубликат (листинг 3.7).

Если перед обработкой с помощью утилиты uniq файл отсортирован, эта утилита обеспечивает отсутствие в файле двух одинаковых строк. (Конечно, утилита sort может справиться с этим и сама, если с ней будет использован ключ `-u`.) Дополнительные сведения можно найти в части V или введя команду `info coreutils 'uniq invocation'`.

diff: Сравнение двух файлов

Утилита diff (difference — разница) проводит сравнение двух файлов и выводит список различий между ними. Эта утилита не изменяет ни один из этих файлов. Ее можно использовать, если надо сравнить два варианта документа или две версии исходного кода программы.

Утилита diff с ключом `-u` (unified output format — унифицированный формат вывода) сначала выводит две строки, показывающие, какой из сравниваемых файлов будет помечен знаком плюс (+), а какой знаком минус (-). В листинге 3.8 знак минус служит признаком файла colors.1, а знак плюс служит признаком файла colors.2.

Листинг 3.7. Утилита uniq, удаляющая строки-дубликаты

```
$ cat dups
Cathy
Fred
Joe
John
Mary
Mary
Paula
$ uniq dups
Cathy
Fred
Joe
John
Mary
Paula
```

Команда `diff -u` разбивает длинный, многострочный текст на *куски* (hunk). Каждый такой кусок предваряется строкой, которая начинается и заканчивается двумя символами at (@@). Эта строка является идентификатором куска, который показывает номер начальной строки и количество строк из каждого файла, попавших в этот кусок. В листинге 3.8 кусок включает часть файла colors.1 (помеченного знаком минус) с первой по шестую строку. Затем значение `+1,5` показывает, что кусок включает также часть файла colors.2 с первой по пятую строку.

После строк заголовка команда `diff -u` показывает каждую строчку текста с ведущим знаком минус, ведущим знаком плюс или пробелом. Знак минус в начале строчки показывает, что строка встречается только в файле, помеченном знаком минус. Знак плюс там же показывает, что строка встречается только в файле, помеченном знаком плюс. Строка, начинающаяся с пробела (то есть ни со знака плюс, ни со знака минус), встречается в обоих файлах в одном и том же месте. Дополнительные сведения можно найти в части V или в info-странице, посвященной утилите diff.

Листинг 3.8. Утилита diff, отображающая унифицированный формат вывода

```
$ diff -u colors.1 colors.2
--- colors.1    2009-07-29 16:41:11.000000000 -0700
+++ colors.2    2009-07-29 16:41:17.000000000 -0700
@@ -1,6 +1,5 @@
red
+blue
green
yellow
-pink
-purple
orange
```

file: Определение содержимого файла

Утилиту file можно использовать для изучения содержимого файла без его открытия и просмотра самого файла. В следующем примере утилита file выдает отчет о том, что

в файле letter_e.bz2 содержатся данные, сжатые утилитой bzip2, которая будет рассмотрена ниже:

```
$ file letter_e.bz2
letter_e.bz2: bzip2 compressed data, block size = 900k
```

Затем утилита file выдает отчет еще о двух файлах:

```
$ file memo zach.jpg
memo:      ASCII text
zach.jpg: JPEG image data, … resolution (DPI), 72 x 72
```

Дополнительные сведения об утилите file можно найти в части V или на соответствующей man-странице.

| (канал): Организация связи между процессами

Поскольку каналы являются неотъемлемой частью функционирования системы Linux, в этой главе они используются в примерах. Подробное описание каналов начинается в разделе «Каналы» на с. 151.

Процессом называется выполнение команды Linux-системой (см. раздел «Процессы» на с. 321). Связь между процессами является характерным признаком систем UNIX и Linux. Канал (отображаемый в командной строке в виде вертикальной черты [|] и обозначаемый на клавиатуре в виде сплошной или прерывистой вертикальной линии) предоставляет самую простую форму этой связи. После простой установки в командную строку канал берет выходную информацию у одной утилиты и отправляет ее в качестве входной информации другой утилите. Если использовать терминологию UNIX и Linux, канал забирает стандартный вывод одного процесса и перенаправляет его, превращая в стандартный ввод другого процесса. (Дополнительные сведения даны в разделе «Стандартный ввод и стандартный вывод» на с. 143.) Основная часть информации, отображаемой процессом на экране, отправляется на стандартное устройство вывода. Если эта информация не перенаправлена, этот вывод появляется на экране. С помощью канала можно перенаправить стандартный вывод, превращая его в стандартный ввод для другой утилиты. Например, утилита head может получать входную информацию из файла, чье имя указано в командной строке после слова head, или же из стандартного ввода. Следующая команда предназначена для сортировки строк файла months (см. листинг 3.5) и использует утилиту head для вывода первых четырех месяцев в виде отсортированного списка:

```
$ sort months | head -4
Apr
Aug
Dec
Feb
```

Следующая командная строка предназначена для отображения количества файлов, имеющихся в каталоге (предположим, что имена файлов не содержат пробелов). Утилита wc (word count — счетчик слов) с ключом -w (words — слова) выводит количество слов в свой стандартный вывод или в файл, указанный в командной строке:

```
$ ls | wc -w
14
```

Можно воспользоваться каналом, чтобы отправить выходную информацию программы на принтер:

```
$ tail months | lpr
```

Еще четыре утилиты

Утилиты echo и date относятся к еще двум часто используемым утилитам из обширной коллекции Linux-утилит. Утилита script записывает часть рабочего сеанса в файл, а утилита todos или unix2dos создает копию текстового файла, которая может быть прочитана на машинах, работающих под управлением Windows или Mac OS X.

echo: Отображение текста

Утилита echo копирует на экран те символы, которые вы ввели в командную строку после команды echo. В листинге 3.9 показано несколько примеров ее использования. В последнем из них показано, как оболочка воспринимает не заключенную в кавычки звездочку (*), введенную в командную строку: она превращает звездочку в список имен файлов, имеющихся в каталоге.

Утилита echo идеально подходит для изучения оболочки и других Linux-утилит. В некоторых примерах, приведенных в разделе «Специальный символ „*“» на с. 158, утилита echo используется для иллюстрации действия специальных символов. В главах 4, 8 и 10 утилита echo помогает объяснить, как работают переменные оболочки и как можно отправлять сообщения из сценариев оболочки на экран.

Листинг 3.9. Утилита echo копирует командную строку (кроме слова echo) на экран

```
$ ls
memo  memo.0714  practice
$ echo Hi
Hi
$ echo This is a sentence.
This is a sentence.
$ echo star: *
star: memo memo.0714 practice
$
```

Дополнительные сведения об этой утилите можно найти в части V или введя команду info coreutils 'echo invocation'. Информацию о версиях утилиты echo, встроенных в оболочки bash и tcsh, можно найти на соответствующих man-страницах.

Утилиту echo можно использовать для создания файла, перенаправив ее выход в файл:

```
$ echo 'My new file.' > myfile
$ cat myfile
My new file.
```

Знак «больше, чем» (>) заставляет оболочку отправить выходную информацию утилиты echo вместо экрана в файл по имени myfile. Дополнительные сведения даны в разделе «Перенаправление стандартного вывода» на с. 146.

date: Отображение даты и времени

Утилита date выводит текущую дату и время:

```
$ date
Wed Mar 18 17:12:20 PDT 2009
```

Команда, приводимая в следующем примере, показывает, как можно изменить формат и выбрать содержимое при выводе даты:

```
$ date +"%A %B %d"
Wednesday March 18
```

Дополнительные сведения можно найти в части V или введя команду `info coreutils 'date invocation'`.

script: Запись сеанса работы с оболочкой

Утилита script записывает весь рабочий сеанс, начиная со входа в систему, включая ваш ввод и ответы системы. Эта утилита может принести пользу только на устройствах, работающих в символьном режиме, таких как терминал или эмулятор терминала. Эта утилита фиксирует сеанс работы с редактором vim, но, поскольку vim использует управляющие символы для позиционирования курсора и отображает разные гарнитуры шрифта, например полужирный шрифт, выходную информацию будет трудно читать, и она может не принести достаточной пользы.

При выводе с помощью утилиты cat содержимого файла, зафиксировавшего сеанс работы с vim, содержимое сеанса быстро промелькнет перед вашими глазами.

По умолчанию утилита script записывает перехваченный сеанс в файл по имени typescript. Чтобы указать другое имя файла, после команды script нужно ввести пробел и имя файла. Чтобы добавить информацию к файлу, после команды script, но до указания имени файла, ставится ключ `-a`. Следующий пример представляет собой сеанс, записанный с помощью утилиты script:

```
$ script
Script started, file is typescript
$ whoami
sam
$ ls -l /bin | head -5
total 5024
-rwxr-xr-x 1 root root    2928 Sep 21 21:42 archdetect
-rwxr-xr-x 1 root root    1054 Apr 26 15:37 autopartition
-rwxr-xr-x 1 root root    7168 Sep 21 19:18 autopartition-loop
-rwxr-xr-x 1 root root  701008 Aug 27 02:41 bash
$ exit
exit
Script done, file is typescript
```

Чтобы остановить запись сеанса, проводимую с помощью утилиты script, используется утилита exit. После этого можно просмотреть созданный файл, используя утилиты cat, less, more или текстовый редактор. В следующем примере показан файл, созданный предыдущей командой script:

```
$ cat typescript
Script started on Thu Sep 24 20:54:59 2009
```

```
$ whoami
Sam
$ ls -l /bin | head -5
total 5024
-rwxr-xr-x 1 root root    2928 Sep 21 21:42 archdetect
-rwxr-xr-x 1 root root    1054 Apr 26 15:37 autopartition
-rwxr-xr-x 1 root root    7168 Sep 21 19:18 autopartition-loop
-rwxr-xr-x 1 root root  701008 Aug 27 02:41 bash
$ exit
exit
Script done on Thu Sep 24 20:55:29 2009
```

Если вы будете редактировать файл с помощью vim, emacs или какого-нибудь другого редактора, можно воспользоваться утилитами fromdos или dos2unix (которые рассматриваются ниже), чтобы удалить из файла символы ^M, которые появляются в конце строк. Дополнительные сведения можно найти на man-странице, посвященной утилите script.

todos/unix2dos: Преобразование файлов Linux и Mac OS X в формат Windows

Если вам нужно использовать текстовый файл, созданный Linux-системой, совместно с кем-нибудь, кто пользуется Windows или Mac OS X, следует преобразовать этот файл перед тем, как пользователь другой системы сможет его прочесть. Утилита todos (to DOS — в формат DOS, входящая в пакет tofrodos) или утилита unix2dos (UNIX to DOS — из UNIX в DOS, входящая в пакет unix2dos) конвертирует текстовый файл Linux, позволяя прочитать его на машине, работающей под Windows или OS X. Чтобы преобразовать файл по имени memo.txt (созданный в тестовом редакторе) в файл DOS-формата, введите следующую команду:

```
$ todos memo.txt
```

или

```
$ unix2dos memo.txt
```

Теперь файл можно отправить по электронной почте в виде вложения какому-нибудь пользователю Windows или OS X. Если не пользоваться никакими ключами, утилита todos перезаписывает исходный файл. Чтобы заставить todos перед изменением файла создать его копию с именем, имеющим расширение .bak, нужно использовать ключ -b (backup — резервная копия). Чтобы заставить утилиту unix2dos записать измененную версию в другой файл, указанный в качестве второго аргумента (unix2dos *старый новый*), используется ключ -n (new — новый).

Утилиты fromdos и dos2unix

Чтобы преобразовать файлы Windows или OS X в формат, доступный для чтения в системе Linux, используются утилиты fromdos (from DOS — из DOS, входящей в пакет tofrodos) или dos2unix (DOS to UNIX — из DOS в UNIX, входящей в пакет dos2unix):

```
$ fromdos memo.txt
```

или

```
$ dos2unix memo.txt
```

Дополнительные сведения можно получить на man-страницах, посвященных утилитам todos, fromdos, unix2dos и dos2unix.

Можно также воспользоваться утилитой tr (translate — перевести, см. часть V) для превращения текстового файла Windows или OS X в текстовый файл Linux. В следующем примере ключ -d (delete — удалить) заставляет утилиту tr удалить символы ввода (представленные сочетанием символов \r) при создании копии файла:

```
$ cat memo | tr -d '\r' > memo.txt
```

Символ «больше, чем» (>) приводит к перенаправлению стандартного вывода утилиты tr в файл по имени memo.txt. Дополнительные сведения можно найти в главе 5, в разделе «Перенаправление стандартного вывода». Преобразование файла другим способом, без использования утилиты todos или утилиты unix2dos, дается труднее.

Сжатие и архивирование файлов

Длинные файлы занимают много дискового пространства и требуют большего времени для передачи их по сети из одной системы в другую. Если содержимое большого файла не требует частого просмотра, может появиться желание сохранить его на компакт-диске, диске DVD или на другом носителе и удалить его с жесткого диска. Если возникает постоянная потребность в файле, то извлечение копии с другого носителя может принести существенные неудобства. Чтобы сократить пространство, занимаемое файлом, можно сжать файл без потери содержащейся в нем информации. К тому же с единым архивом, содержащим сразу несколько файлов, запакованных в один большой файл, проще работать. Например, его проще выкладывать в сеть и рассылать по электронной почте. Сжатые, заархивированные файлы можно также загрузить из Интернета. Утилиты, рассмотренные в этом разделе, сжимают файлы и возвращают их в обычное состояние, а также запаковывают файлы в архивы и распаковывают их из этих архивов.

bzip2: Сжатие файла

Утилита bzip2 сжимает файл путем анализа его содержимого и более рациональной записи. Новая версия файла выглядит совершенно по-другому. Фактически из-за наличия в новом файле множества невыводимых на печать символов его нельзя просматривать непосредственно. Утилита bzip2 особенно хорошо работает с файлами, в которых содержится много повторяющейся информации. К такой информации относятся текстовые и графические данные, хотя многие изображения уже находятся в сжатом формате.

В следующем примере показан довольно скучный файл. Каждые 8000 строк файла letter_e содержат по 72 символа e и символ новой строки, которым отмечен конец строки. Файл занимает более полумегабайта дискового пространства.

```
$ ls -l
-rw-rw-r--  1 sam sam 584000 Mar  1 22:31 letter_e
```

Ключ -l (long — развернутый формат) заставляет утилиту ls вывести дополнительную информацию о файле. В этом примере утилита показывает, что файл letter_e имеет

длину 584 000 байт. Ключ –v (verbose — подробно) заставляет утилиту bzip2 вывести отчет о том, насколько можно сократить размер файла. В данном случае утилита сокращает файл на 99,99 процента:

```
$ bzip2 -v letter_e
letter_e: 11680.00:1, 0.001 bits/byte, 99.99% saved, 584000 in, 50 out.
$ ls -l
-rw-rw-r--  1 sam sam 50 Mar  1 22:31 letter_e.bz2
```

Расширение имени файла .bz2

Теперь длина файла составляет всего лишь 50 байтов. Утилита bzip2 также изменяет имя файла, добавляя к нему расширение .bz2. Это напоминает о том, что файл находится в сжатом состоянии; для того чтобы отобразить или вывести на печать такой файл, его сначала следует распаковать. Утилита bzip2 не изменяет связанную с файлом дату модификации, даже если она полностью изменяет содержимое файлов.

СОХРАНЯЙТЕ ИСХОДНЫЙ ФАЙЛ С ПОМОЩЬЮ КЛЮЧА –K

Утилита bzip2 и ее антипод, утилита bunzip2, удаляют исходный файл при его сжатии или распаковке. Использование ключа –k (keep — сохранить) позволяет сохранить исходный файл.

В следующем, более полезном примере файл zach.jpg содержит изображение компьютерной графики:

```
$ ls -l
-rw-r--r--  1 sam sam 33287 Mar  1 22:40 zach.jpg
```

Утилита bzip2 способна уменьшить размер файла всего на 28 процентов, поскольку изображение уже находится в сжатом формате:

```
$ bzip2 -v zach.jpg
zach.jpg:  1.391:1,  5.749 bits/byte, 28.13% saved, 33287 in, 23922 out.
$ ls -l
-rw-r--r--  1 sam sam 23922 Mar  1 22:40 zach.jpg.bz2
```

Дополнительные сведения можно найти в части V и на man-странице, посвященной утилите bzip2. Можно также зайти на веб-страницу www.bzip.org и обратиться к документу Bzip2 mini-HOWTO (инструкции, как получить этот документ, были даны в разделе «Документы HOWTO: Информация о том, как все работает» на с. 58).

bunzip2 и bzcat: Распаковка файла

Для восстановления прежнего состояния файла, сжатого с помощью утилиты bzip2, используется утилита bunzip2:

```
$ bunzip2 letter_e.bz2
$ ls -l
-rw-rw-r--  1 sam sam 584000 Mar  1 22:31 letter_e
$ bunzip2 zach.jpg.bz2
$ ls -l
-rw-r--r--  1 sam sam  33287 Mar  1 22:40 zach.jpg
```

Утилита bzcat отображает содержимое файла, сжатого с помощью bzip2. Являясь эквивалентом cat для файлов с расширением .bz2, утилита bzcat распаковывает сжатые данные и отображает их уже распакованными. Как и cat, утилита bzcat не изменяет

содержимое исходного файла. Канал, использованный в следующем примере, перенаправляет выходную информацию bzcat, которая становится входными данными для утилиты head, выводящей на экран первые две строки файла:

```
$ bzcat letter_e.bz2 | head -2
eeee
```

При создании архива утилита tar добавляет к нему избыточные данные. В следующем примере архивный файл all.tar занимает около 9700 байт, а сумма размеров трех файлов составляет около 6000 байт. Эти издержки более заметны на небольших файлах.

```
$ ls -l g b d
-rw-r--r--   1 zach  other 1178 Aug 20 14:16 b
-rw-r--r--   1 zach  zach  3783 Aug 20 14:17 d
-rw-r--r--   1 zach  zach  1302 Aug 20 14:16 g
$ tar -cvf all.tar g b d
g
b
d
$ ls -l all.tar
-rw-r--r--   1 zach      zach         9728 Aug 20 14:17 all.tar
$ tar -tvf all.tar
-rw-r--r-- zach /zach     1302 2009-08-20 14:16 g
-rw-r--r-- zach /other    1178 2009-08-20 14:16 b
-rw-r--r-- zach /zach     3783 2009-08-20 14:17 d
```

Последняя команда в предыдущем примере использует ключ -t для отображения таблицы содержимого этого архива. Использование ключа -x вместо ключа -t приведет к извлечению файлов из tar-архива. Если нужно, чтобы утилита tar делала свою работу молча[1], следует оказаться от ключа -v.

Сжатые tar-файлы

Для сжатия tar-файлов, позволяющего облегчить их хранение и работу с ними, можно воспользоваться утилитами bzip2, compress или gzip. Многие файлы, загружаемые из Интернета, сразу находятся в одном из этих форматов. Для файлов, обработанных утилитой tar и сжатых утилитой bzip2, часто используется расширение .tar.bz2 или .tbz. Файлы, обработанные утилитами tar и gzip, имеют расширение .tar.gz, .tgz или .gz, а файлы, обработанные утилитами tar и compress, используют в качестве расширения .tar.Z.

Распаковать файлы, обработанные утилитами tar и gzip, можно в два этапа. (Для файла, сжатого утилитой bzip2, нужно применить такую же процедуру, но при этом вместо gunzip следует использовать утилиту bunzip2.) В следующем примере показано, как можно распаковать GNU-утилиту make после ее загрузки (ftp.gnu.org/pub/gnu/make/make-3.81.tar.gz):

```
$ ls -l mak*
-rw-r--r-- 1 sam sam 1564560 Mar 26 18:25 make-3.81.tar.gz
$ gunzip mak*
$ ls -l mak*
-rw-r--r-- 1 sam sam 6072320 Mar 26 18:25 make-3.81.tar
$ tar -xvf mak*
make-3.81/
make-3.81/config/
```

[1] Хотя исходная UNIX-версия утилиты tar не использует в ключах лидирующего дефиса, обозначающего их присутствие в командной строке, GNU- и Linux-версии воспринимают дефисы, но прекрасно работают и без них. В данной книге дефисы перед ключами утилиты tar ставятся, чтобы сохранить однообразие с порядком использования большинства других утилит.

```
make-3.81/config/dospaths.m4
make-3.81/config/gettext.m4
make-3.81/config/iconv.m4
...
make-3.81/tests/run_make_tests
make-3.81/tests/run_make_tests.pl
make-3.81/tests/test_driver.pl
```

Первая команда выводит сведения о загруженном файле, который был обработан утилитами tar и gzip: make-3.81.tar.gz (его размер составляет около 1,2 Мб). Звездочка (*) в имени файла соответствует любым символам в любых именах файлов (см. раздел «Специальный символ „*“» на с. 158, поэтому ls отображает список файлов, имена которых начинаются с «mak». В данном случае есть только один такой файл. Использование звездочки экономит время на набор текста и может уберечь от ошибок, допускаемых при наборе длинных имен. Команда gunzip вызывает распаковку файла и выдает файл make-3.81.tar (в имени которого уже отсутствует расширение .gz), размер которого составляет около 4,8 Мб. По команде tar в рабочем каталоге создается каталог make-3.81, и в него происходит распаковка файлов.

```
$ ls -ld mak*
drwxr-xr-x 8 sam sam    4096 Mar 31  2006 make-3.81
-rw-r--r-- 1 sam sam 6072320 Mar 26 18:25 make-3.81.tar
$ ls -l make-3.81
total 2100
-rw-r--r-- 1 sam sam  53838 Mar 31  2006 ABOUT-NLS
-rw-r--r-- 1 sam sam   2810 Mar 19  2006 AUTHORS
-rw-r--r-- 1 sam sam  18043 Dec 10  1996 COPYING
-rw-r--r-- 1 sam sam 106390 Mar 31  2006 ChangeLog
...
-rw-r--r-- 1 sam sam  16907 Feb 11  2006 vmsjobs.c
-rw-r--r-- 1 sam sam  17397 Feb 11  2006 vpath.c
drwxr-xr-x 6 sam sam   4096 Mar 31  2006 w32
```

После того как tar извлечет файлы из архива, в рабочем каталоге будут находиться два файла с именами, начинающимися на «mak»: make-3.81.tar и make-3.81. Ключ -d (directory — каталог) заставляет утилиту ls отобразить только имена файлов и каталогов и не выводить содержимое каталогов, как она это делает в обычном режиме. Выполнение последней команды ls приводит к отображению файлов и каталогов, содержащихся в каталоге make-3.81. Дополнительные сведения можно найти в части V и на man-странице, посвященной утилите tar.

TAR: КЛЮЧ –X МОЖЕТ ПРИВЕСТИ К ИЗВЛЕЧЕНИЮ БОЛЬШОГО КОЛИЧЕСТВА ФАЙЛОВ

Некоторые tar-архивы содержат большое количество файлов. Чтобы вывести список файлов, содержащихся в архиве без их распаковки, следует запустить утилиту tar с ключом -t и с именем tar-файла. В некоторых случаях может понадобиться создать новый каталог (см. описание утилиты mkdir в главе 4), переместить tar-файл в этот каталог и там же его распаковать. Благодаря этому распакованные файлы не будут смешиваться с уже существующими файлами, не создавая при этом путаницу. Этот прием позволяет также упростить удаление извлеченных из архива файлов. В зависимости от того, каким образом они были созданы, некоторые tar-айлы автоматически создают новый каталог и помещают в него файлы, извлекаемые из архива. Ключ -t показывает, куда утилита tar будет помещать извлекаемые из архива файлы.

TAR: КЛЮЧ –X МОЖЕТ ПРИВЕСТИ К ПЕРЕЗАПИСИ ФАЙЛОВ

Применение ключа –x при запуске утилиты tar приводит к перезаписи файла, имеющего такое же имя, как и у извлекаемого файла. Чтобы избежать перезаписи файлов, последуйте совету из предыдущего предупреждения.

Вызов утилит gunzip и tar можно объединить в одной командной строке с помощью канала (|), который перенаправит выходные данные утилиты gunzip, сделав их входными данными для утилиты tar:

```
$ gunzip -c make-3.81.tar.gz | tar -xvf -
```

Ключ –c заставляет утилиту gunzip вместо создания файла отправить ее выходные данные через канал. Последний дефис (–) заставляет утилиту tar считывать данные из стандартного ввода. Дополнительную информацию о том, как работает эта командная строка, можно найти в разделе «Каналы» на с. 151, а также в данной главе и в части V (утилиты gzip и tar). Более простым решением будет применение с утилитой tar ключа –z. Этот ключ заставляет tar вызвать утилиту gunzip (или gzip, когда вы создаете архив) напрямую и упрощает предыдущую командную строку до следующего вида:

```
$ tar -xvzf make-3.81.tar.gz
```

Точно так же применение ключа –j приводит к вызову утилиты bzip2 или утилиты bunzip2.

Команды для поиска мест, где находятся утилиты

Утилиты whereis и slocate помогают в поиске нужной утилиты, чье имя вы забыли, или той утилиты, про которую вы не знаете, где она находится. Когда есть несколько копий утилиты или программы, утилита which может подсказать вам, какую копию вы будете запускать. Утилита slocate ищет файлы на локальной системе.

which и whereis: Определение местоположения утилиты

Путь поиска

Когда системе Linux выдается какая-нибудь команда, оболочка просматривает список каталогов в поиске программы с таким именем и запускает первую найденную программу. Список каталогов называется *путем поиска*. Информация о том, как изменить путь поиска, изложена в разделе «Переменная PATH: Указание тех мест, где оболочка ищет программы» на с. 312. Если вы не изменили путь поиска, оболочка просматривает только стандартный набор каталогов, после чего прекращает поиск. Однако полезные утилиты могут находиться и в других каталогах системы.

Утилита which

Утилита which определяет места, где находятся утилиты, отображая полный путь к имени файла утилиты. (В главе 4 имеются дополнительные сведения по структуре файловой системы Linux.) Локальная система может включать несколько утилит с одинаковыми именами. При вводе имени утилиты оболочка ищет утилиту в вашем пути поиска и запускает первую попавшуюся утилиту с таким именем. Утилита which помогает определить, какая копия утилиты будет запущена оболочкой. В следующем примере which выводит отчет о местоположении утилиты tar:

```
$ which tar
/bin/tar
```

Утилита which может пригодиться в том случае, когда возникают подозрения, что утилита не работает должным образом. Воспользовавшись утилитой which, можно обнаружить, что запускается нестандартная версия инструментального средства или версия, отличающаяся от той, запуск которой ожидался. (Сведения о стандартном размещении исполняемых файлов даны в разделе «Важные стандартные каталоги и файлы» на с. 111.) Если, к примеру, утилита tar не работает должным образом и обнаруживается, что вместо /bin/tar вы запускаете /usr/local/bin/tar, можно предположить, что эта локальная версия повреждена.

Утилита whereis

Утилита whereis ищет файлы, имеющие отношение к указанной утилите, осуществляя поиск не в вашем пути поиска, а в стандартных местах. Например, вы можете найти места, где находятся файлы, имеющие отношение к утилите tar:

```
$ whereis tar
tar: /bin/tar /usr/include/tar.h /usr/share/man/man1/tar.1.gz
```

В данном примере утилита whereis находит три файла, которые имеют отношение к утилите tar: файл утилиты tar, заголовочный файл tar и man-страницу, посвященную tar.

СРАВНЕНИЕ УТИЛИТ WHICH И WHEREIS

В соответствии со своим именем, утилита which («которая») ведет по очереди поиск по каталогам, включенным в ваш путь поиска (см. на с. 312, раздел «Переменная PATH: Указание тех мест, где оболочка ищет программы», и находит указанную утилиту. Если в вашем пути поиска содержится более одной утилиты с указанным именем, утилита which отображает имя только первой из них (той, которая будет запущена при выполнении команды). Утилита whereis ищет в стандартных каталогах, работая независимо от вашего пути поиска. Утилиту whereis следует использовать для определения местонахождения бинарного (исполняемого) файла, а также любых страниц руководства и исходного кода для указанной вами программы; утилита whereis отображает все найденные ею файлы.

КОМАНДЫ WHICH, WHEREIS И ВСТРОЕННЫЕ КОМАНДЫ

Обе утилиты, и which, и whereis, выдают отчет только об именах утилит, в том виде, в котором они присутствуют на диске, и не дают сведений об утилитах, встроенных в оболочку (см. главу 5, раздел «Встроенные команды»). Когда вы используете whereis, пытаясь найти место, где находится утилита echo (которая существует как в виде служебной программы, так и в виде утилиты, встроенной в оболочку), получается следующий результат:

```
$ whereis echo
echo: /bin/echo /usr/share/man/man1/echo.1.gz
```

Утилита whereis не отображает встроенную версию утилиты echo. Даже утилита which выводит неверную информацию:

```
$ which echo
/bin/echo
```

Работая с оболочкой bash, можно ввести встроенную команду type (см. главу 10, раздел «Встроенные команды»), чтобы определить, относится ли указанная утилита к таким же встроенным командам:

```
$ type echo
echo is a shell builtin (echo является командой, встроенной в оболочку)
```

slocate — locate: Поиск файла

Утилита slocate (secure locate — безопасное определение местонахождения), как и утилита locate, ведет поиск файлов на локальной системе:

```
$ slocate motd
/usr/share/app-install/icons/xmotd.xpm
/usr/share/app-install/desktop/motd-editor.desktop
/usr/share/app-install/desktop/xmotd.desktop
/usr/share/base-files/motd.md5sums
/usr/share/base-files/motd
...
```

Возможно, чтобы воспользоваться утилитами slocate и locate, их придется устанавливать в систему. Перед первым использованием slocate или locate утилита updatedb должна создать или обновить базу данных slocate. Обычно такая база данных обновляется раз в день с помощью сценария cron (сведения о crontab и cron можно найти в части V).

ЕСЛИ ВЫ НЕ ПОДКЛЮЧЕНЫ К СЕТИ, МОЖЕТЕ ПРОПУСТИТЬ ОСТАВШУЮСЯ ЧАСТЬ ДАННОЙ ГЛАВЫ

Если вы не пользуетесь системой, подключенной к сети, можете пропустить все оставшиеся разделы данной главы. Если вы работаете не в сети, но имеете настройки для получения и отправки сообщений электронной почты, прочитайте раздел «Электронная почта».

Получение информации о пользователе и системе

В этом разделе рассматриваются утилиты, предоставляющие информацию о том, кто пользуется системой, чем заняты эти пользователи и как работает система в данный момент.

Чтобы определить, кто пользуется локальной системой, можно воспользоваться одной из нескольких утилит, отличающихся друг от друга предоставляемыми услугами и поддерживаемыми ключами. Самая простая утилита who выводит список пользователей, вошедших в данную локальную систему, список устройств, задействованных каждым пользователем, а также время входа каждого пользователя в систему.

Утилиты w и finger выводят более подробную информацию, например полные имена пользователей и содержимое командной строки, запущенной каждым из них. Если локальная система подключена к сети, утилиту finger можно использовать для извлечения информации о пользователях, работающих на удаленных системах. В табл. 3.1 приведена сводка данных, выводимых этими утилитами.

who: Вывод списка пользователей, вошедших в систему

Утилита who выводит список пользователей, вошедших в локальную систему. В первом столбце листинга 3.10 утилита who показывает, что в систему вошли пользователи Sam, Max и Zach. (Max вошел в систему сразу из двух мест.) Во втором столбце показаны устройства, с которых вошел в систему каждый пользователь. К ним относятся терминалы, рабочие станции или эмуляторы терминалов. В третьем столбце показаны дата и время входа пользователя в систему. В необязательном четвертом столбце показаны (в скобках) имена систем, из которых вошел в систему удаленный пользователь.

Информация, отображаемая утилитой who, может использоваться в том случае, если вы хотите связаться с пользователем, вошедшим в локальную систему. Когда пользователь входит в систему, вы можете с целью немедленной установки связи с ним

воспользоваться утилитой write (см. раздел «Связь с другими пользователями»). Если утилита who не выводит список пользователей или если вам не нужна немедленная установка связи, вы можете послать любому пользователю сообщение по электронной почте (см. раздел «Электронная почта»).

Если информация, выводимая утилитой who, вызывает прокрутку экрана, можно ее перенаправить через канал (|, см. раздел «Каналы» на с. 151), чтобы она стала входной информацией для утилиты less, осуществляющей поэкранное отображение выходных данных. Можно также воспользоваться каналом для перенаправления выходной информации через утилиту grep, чтобы найти указанное имя.

Если нужно определить, каким терминалом вы пользуетесь или в какое время вошли в систему, можно воспользоваться командой who am i:

```
$ who am i
max      tty2         2009-07-25 16:42
```

Листинг 3.10. Утилита who выводит список всех вошедших в систему

```
$ who
sam      tty4         2009-07-25 17:18
max      tty2         2009-07-25 16:42
zach     tty1         2009-07-25 16:39
max      pts/4        2009-07-25 17:27 (coffee)
```

Листинг 3.11. Утилита finger, первый пример: показывает, кто работает в системе

```
$ finger
Login    Name           Tty      Idle  Login Time   Office …
max      Max Wild      *tty2           Jul 25 16:42
max      Max Wild       pts/4      3  Jul 25 17:27 (coffee)
sam      Sam the Great *tty4      29  Jul 25 17:18
zach     Zach Brill    *tty1     1:07  Jul 25 16:39
```

finger: Вывод списка пользователей, вошедших в систему

Утилиту finger можно использовать для вывода списка всех пользователей, вошедших в локальную систему. Дополнительно к именам пользователей утилита finger предоставляет их полные имена, а также сведения о тех устройствах, к которым подключены пользовательские терминалы, время, прошедшее с тех пор, когда пользователь в последний раз вводил что-нибудь с клавиатуры, когда он вошел в систему, и доступную контактную информацию. Если пользователь вошел в систему по сети, то в качестве места, где находится пользователь, показывается имя удаленной системы. Например, в листинге 3.11 пользователь Max вошел в систему с удаленной системы с именем coffee. Звездочка (*), поставленная перед именем устройства в столбце Tty, свидетельствует о том, что пользователь заблокировал сообщения, посылаемые непосредственно на его терминал (см. раздел «mesg: Отказ в приеме или прием сообщений» на с. 92).

УТИЛИТА FINGER МОЖЕТ СОЗДАТЬ ОПАСНУЮ СИТУАЦИЮ

На системах, имеющих повышенные требования к безопасности, системные администраторы могут отключить утилиту finger. Эта утилита может показать информацию, помогающую злоумышленникам взломать систему.

В Mac OS X поддержка удаленного вызова утилиты finger отключена по умолчанию.

Утилиту finger можно также использовать для изучения дополнительной информации об отдельном пользователе, указав имя пользователя в командной строке. В листинге 3.12 утилита finger выводит подробную информацию о пользователе по имени Max, который вошел в систему и активно пользуется одним из своих терминалов (tty2). Другим своим терминалом (pts/4) он не пользовался в течение 3 минут и 7 секунд. Утилита finger может также предоставить сведения о том, что назначить встречу с Max можно в том случае, если связаться с пользователем Zach по дополнительному номеру 1693.

Файлы .plan и .project

Основная информация в листинге 3.12 была собрана утилитой finger из системных файлов. А информация, показанная после заголовка Plan:, была предоставлена самим пользователем Max. Утилита finger ищет файл по имени .plan в личном каталоге пользователя Max и отображает его содержимое.

Листинг 3.12. Утилита finger, второй пример: выводит подробный список, касающийся только одного пользователя

```
$ finger max
Login: max                              Name: Max Wild
Directory: /home/max                    Shell: /bin/tcsh
On since Fri Jul 25 16:42 (PDT) on tty2 (messages off)
On since Fri Jul 25 17:27 (PDT) on pts/4 from coffee
   3 minutes 7 seconds idle
New mail received Sat Jul 25 17:16 2009 (PDT)
     Unread since Sat Jul 25 16:44 2009 (PDT)
Plan:
I will be at a conference in Hawaii all next week.
If you need to see me, contact Zach Brill, x1693.
План:
Всю следующую неделю я буду на конференции на Гавайях.
Если хотите встретиться со мной, сообщите об этом Заку Брилю по доп.номеру 1693.
```

(Имена файлов, начинающиеся с точки, например .plan, обычно не показываются в списке, выводимом утилитой ls, и называются скрытыми именами файлов.) Возможно, вы тоже посчитаете полезным создание своего собственного файла .plan, в котором может содержаться любая информация по вашему выбору, например ваши планы, интересы, номер телефона или адрес. Точно таким же образом утилита finger отображает содержимое файлов .project и .pgpkey, которые находятся в вашем личном каталоге. Если пользователь Max не вошел в систему, утилита finger выведет только его пользовательскую информацию, последнее время его входа в систему, последнее время чтения им своей электронной почты и его план.

Утилиту finger можно также использовать для отображения пользовательского имени конкретного пользователя. Например, работая в системе, которой пользуется некая Helen Simpson, вы можете знать, что фамилия Helen — Simpson, но можете не догадываться о том, что в качестве имени пользователя он использует сочетание hls. Утилита finger, которая не обладает чувствительностью к регистру букв, может найти информацию о Helen, используя ее настоящее имя или фамилию. Благодаря следующим командам осуществляется успешный поиск нужной информации, а также информации о других пользователях по имени Helen или по фамилии Simpson:

```
$ finger HELEN
Login: hls                              Name: Helen Simpson.
…
$ finger simpson
Login: hls                              Name: Helen Simpson.
…
```

Дополнительные сведения об утилите finger можно найти в части V.

w: Вывод списка пользователей, вошедших в систему

Утилита w выводит список пользователей, вошедших на локальную систему. Для утилиты w справедливо утверждение, которое уже рассматривалось в разделе, посвященном утилите who: информация, выводимая утилитой w, может пригодиться, когда появится желание связаться с кем-нибудь из вашего учреждения.

В первом столбце листинга 3.13 показано, что в систему вошли пользователи Max, Zach и Sam. Во втором столбце показаны имена файлов устройств, к которым подключены терминалы каждого пользователя. В третьем столбце показана система, из которой вошел в локальную систему удаленный пользователь. В пятом столбце показано, как долго бездействует каждый пользователь (точнее, сколько времени прошло, с тех пор как он в последний раз нажимал на клавишу своей клавиатуры). В следующих двух столбцах показано, сколько процессорного времени использовал каждый пользователь за текущий сеанс работы, и запущенная им задача, на которую затрачено это время. В последнем столбце показана команда, запущенная каждым пользователем.

Первая строка, которую выводит утилита w, включает время суток, период времени, который проработал компьютер (в днях, часах и минутах), количество вошедших в систему пользователей и среднюю загруженность системы (степень занятости системы). Три числа, характеризующие среднюю загруженность системы, представляют собой количество заданий, ожидающих запуска, среднее количество которых выведено за последние 1, 5 и 15 минут. Чтобы вывести только эту строку, следует воспользоваться утилитой uptime. Сравнительная характеристика утилит w, who и finger приводится в табл. 3.1.

Листинг 3.13. Утилита w

```
$ w
 17:47:35 up 1 day,  8:10,  4 users,  load average: 0.34, 0.23, 0.26
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
sam      tty4     -                17:18    29:14m  0.20s  0.00s vi memo
max      tty2     -                16:42    0.00s  0.20s  0.07s w
zach     tty1     -                16:39    1:07   0.05s  0.00s run_bdgt
max      pts/4    coffee           17:27    3:10m  0.24s  0.24s -bash
```

Таблица 3.1. Сравнение утилит w, who и finger

Отображаемая информация	w	who	finger
Имя пользователя	X	x	x
Идентификатор терминального канала	X	x	x
Время входа в систему (и день для более отдаленных по времени входов)	X		

Отображаемая информация	w	who	finger
Дата и время входа в систему		x	x
Время простоя	X		x
Программа, выполняемая пользователем	X		
Место, с которого пользователь вошел в систему			x
Затраченное на пользователя процессорное время	X		
Полное имя (или другая информация из /etc/passwd)			x
Собственная информация, предоставленная пользователем			x
Время работы системы и ее средняя загруженность	X		

Связь с другими пользователями

Утилиты, рассматриваемые в этом разделе, можно использовать для обмена сообщениями и файлами с другими пользователями, как в диалоговом режиме, так и по электронной почте.

write: Отправка сообщения

Утилита write отправляет сообщение другому пользователю, вошедшему в систему. Когда вы и другой пользователь используете write для отправки друг другу сообщений, устанавливается двухсторонняя связь. Сначала команда write приводит к появлению баннера на терминале другого пользователя с объявлением о ваших намерениях послать ему сообщение (листинг 3.14).

Для команды write используется следующий синтаксис командной строки:

```
write имя_пользователя [терминал]
```

Листинг 3.14. Утилита write, первый пример

```
$ write max
Hi Max, are you there? o   Привет, Макс, ты на связи? o
```

Листинг 3.15. Утилита write, второй пример

```
$ write max
Hi Max, are you there? o   Привет, Макс, ты на связи? o
Message from max@bravo.example.com on pts/0 at 16:23 ...
Yes Zach, I'm here. o      Да, Зак, я на связи. o
```

Параметр *имя_пользователя* относится к пользователю, с которым нужно связаться. Параметр *терминал* — необязательное имя устройства, которое применяется в том случае, если пользователь вошел в систему более одного раза. С помощью утилит who, w или finger можно вывести имена пользователей и имена устройств, относящиеся ко всем пользователям, вошедшим в локальную систему.

Чтобы установить двухстороннюю связь с другим пользователем, и вы, и этот другой пользователь должны ввести команду write, указав в ней имя пользователя, под которым работает ваш собеседник. После этого утилита write построчно копирует текст с одной клавиатуры-дисплея на другой (листинг 3.15). Иногда она помогает вы-

разить намерения, например ввести o (что значит «over» — переход на прием), когда вы готовы к тому, чтобы другой человек приступил к набору сообщения, или ввести oo (что значит «over and out» — переход на прием и предложение к завершению), когда вы готовы завершить общение. Когда нужно прекратить связь с другим пользователем, следует в начале строки нажать комбинацию CONTROL+D. Эта комбинация заставляет утилиту write прекратить работу, приводит к отображению символа конца файла — EOF (end of file) на терминале другого пользователя и возвращает управление вашей оболочке. Другой пользователь должен сделать то же самое.

Если сообщение баннера появляется на вашем экране и закрывает информацию, с которой вы работаете, следует нажать комбинацию CONTROL+ L или CONTROL+ R, чтобы обновить изображение экрана и удалить баннер.

После этого можно завершить недоделанную работу, выйти в оболочку и ответить человеку, который написал вам сообщение. Но при этом нужно запомнить, кто именно написал сообщение, поскольку баннер на экране больше не появится.

mesg: Отказ в приеме или прием сообщений

По умолчанию вывод сообщений на ваш экран заблокирован. Чтобы разрешить другим пользователям отправлять вам сообщения, нужно ввести следующую команду:

```
$ mesg y
```

Если пользователь Max не ввел этой команды до того, как пользователь Zach попытался отправить ему сообщение, у пользователя Zach может появиться следующее сообщение:

```
$ write max
write: max has messages disabled        У пользователя max выключены сообщения
```

Заблокировать сообщения можно командой mesg n. Ввод одной только команды mesg приведет к выводу символа y (означающего «yes», то есть «Да, сообщения разрешены») или символа n (означающего «no», то есть «Нет, сообщения запрещены»).

Если у вас заблокированы сообщения и вы отправляете сообщение другому пользователю с помощью утилиты write, эта утилита выводит следующее сообщение, поскольку, даже если вам разрешено написать другому пользователю, этот пользователь все равно не сможет вам ответить:

```
$ write max
write: write: you have write permission turned off.
                        У вас нет прав пользоваться утилитой write
```

Электронная почта

Электронная почта позволяет связываться с пользователями на локальной системе, и если установка системы представляет собой часть сети, то и с другими пользователями, находящимися в сети. При подключении к Интернету вы можете установить электронную связь с пользователями по всему миру.

Утилиты электронной почты отличаются от утилиты write тем, что они могут отправлять сообщения, когда их получатель не находится в системе. В таком случае

электронная почта хранится до тех пор, пока получатель ее не прочитает. Эти утилиты могут также отправить одно и то же сообщение сразу нескольким пользователям.

В Linux доступны многие программы электронной почты, включая исходную программу mailx, работающую в символьном режиме, программы Mozilla/Thunderbird, pine, программу Kmail и программы, являющиеся их развитием. Электронную почту можно также отправить и из редактора emacs. Еще одной популярной графической программой электронной почты является sylpheed (sylpheed.sraoss.jp/en).

Есть еще две программы, которые могут упростить работу с любой программой электронной почты и сделать работу с ней более безопасной. Программа procmail (www.procmail.org) создает и обслуживает серверы электронной почты и списки рассылки, проводит предварительную обработку электронной почты, сортируя ее в соответствующие файлы и каталоги; запускает различные программы, в зависимости от характеристик входящих почтовых сообщений; пересылает сообщения дальше и т. д. Программа защиты конфиденциальности — GNU Privacy Guard (GPG или GNUpg; www.gnupg.org) занимается шифрованием и дешифрованием сообщений электронной почты, делая их практически недоступными для чтения посторонними лицами.

Сетевые адреса

Если локальная система является частью локальной сети, у вас, как правило, есть возможность отправлять сообщения электронной почты другим пользователям, работающим в других подключенных к той же сети системах, и получать от них такие же сообщения. Тот, кто отправляет сообщение электронной почты пользователю Max через Интернет, должен наряду с именем пользователя указать его *доменное имя*. Чтобы отправить сообщение по электронной почте автору этой книги, нужно воспользоваться адресом mgs@sobell.com.

Заключение

Утилиты, представленные в данной главе, являются не очень большой, но довольно мощной выборкой утилит, доступных в обычной Linux-системе. Поскольку все они относятся к наиболее востребованным утилитам и являются неотъемлемой частью материала следующих глав, важно научиться свободно ими пользоваться.

Утилиты, список которых приведен в табл. 3.2, предназначены для работы с файлами, в том числе для отображения, сравнения и вывода на печать их содержимого.

Таблица 3.2. Утилиты для работы с файлами

Утилита	Функция
cp	Копирует один или несколько файлов
diff	Показывает разницу между двумя файлами
file	Выводит информацию о содержимом файла
grep	Ищет строку в файле (файлах)
head	Выводит строки из начала файла
lpq	Выводит список заданий, имеющихся в очереди на печать
lpr	Ставит файл (файлы) в очередь на печать
lprm	Удаляет задание из очереди на печать

продолжение ⇩

Таблица 3.2 (*продолжение*)

Утилита	Функция
mv	Осуществляет переименование файла или перемещает файл(ы) в другой каталог
sort	Построчно выстраивает содержимое файла в определенном порядке
tail	Выводит строки с конца файла
uniq	Выводит содержимое файла, пропуская смежные строки-дубликаты

Чтобы сократить объем дискового пространства, занимаемого файлом, этот файл можно сжать с помощью утилиты bzip2. Сжатию особенно хорошо поддаются файлы, содержащие шаблоны, а также большинство текстовых файлов, но эта операция сокращает размер практически любых файлов. Утилита, обратная по действию bzip2 — bunzip2, восстанавливает файл в его первозданном (несжатом) виде. В табл. 3.3 приводится список утилит, которые сжимают файлы и возвращают их в обычное состояние. Наибольшей эффективностью при этом обладает утилита bzip2.

Таблица 3.3. Утилиты для сжатия и возвращения файлов в обычное состояние

Утилита	Функция
bunzip2	Возвращает файл, сжатый утилитой bzip2, в его обычном размере и формате
bzcat	Отображает содержимое файла, сжатого утилитой bzip2
bzip2	Сжимает файл
compress	Сжимает файл
gunzip	Возвращает файл, сжатый утилитой gzip или утилитой compress, в его обычном размере и формате
gzip	Сжимает файл (но менее эффективно, чем утилита bzip2)
zcat	Отображает содержимое файла, сжатого утилитой gzip

Архив — это файл, зачастую подвергшийся сжатию, который содержит группу файлов. Утилита tar (табл. 3.4) занимается запаковкой и распаковкой архивов. Расширения имен файлов .tar.bz2, .tar.gz и .tgz показывают, что это сжатые, архивные файлы, созданные утилитой tar, которые часто встречаются в программных пакетах, получаемых из Интернета.

Таблица 3.4. Утилита для архивирования файлов

Утилита	Функция
tar	Создает архивный файл или извлекает из него файлы

Утилиты, список которых приведен в табл. 3.5, позволяют определить, где находится утилита на локальной системе. Например, они могут вывести путевое имя утилиты или список компиляторов C++, доступных в системе.

Таблица 3.5. Утилиты для определения местонахождения утилит

Утилита	Функция
locate	Ищет файлы на локальной системе
whereis	Выводит полные путевые имена утилиты, исходного кода или man-страницы
which	Выводит полное путевое имя запущенной вами утилиты

В табл. 3.6 приводится список утилит, отображающих информацию о других пользователях. Можно без труда узнать полное имя пользователя, состояние его входа в систему, название оболочки, из которой он вошел в систему, и другие сведения, поддерживаемые системой.

Таблица 3.6. Утилиты для получения информации о пользователях и системе

Утилита	Функция
finger	Выводит подробные сведения о пользователях, включая их полные имена
hostname	Выводит имя локальной системы
w	Выводит подробные сведения о пользователях, вошедших в локальную систему
who	Выводит сведения о пользователях, вошедших в локальную систему

Утилиты, показанные в табл. 3.7, могут помочь вам поддерживать отношения с другими пользователями локальной сети.

Таблица 3.7. Утилиты для связи пользователей друг с другом

Утилита	Функция
mesg	Разрешает или запрещает сообщения, отправляемые утилитой write
write	Отправляет сообщение другому пользователю, вошедшему в систему

В табл. 3.8 приводятся утилиты разного назначения

Таблица 3.8. Утилиты разного назначения

Утилита	Функция
date	Выводит текущую дату и время
echo	Копирует свои аргументы на экран

Упражнения

1. Какую команду можно использовать, чтобы определить, кто вошел в систему с определенного терминала?
2. Как можно воспрепятствовать использованию утилиты write другими пользователями для связи с вами? Зачем это может понадобиться?
3. Что происходит, когда вы вводите следующие команды, если файл по имени done уже существует?

```
$ cp to_do done
$ mv to_do done
```

4. Как можно вывести следующую строку на ваш рабочий экран?

```
Hi $2, I'm number * (guess)!
```

5. Как найти номер телефона Ace Electronics в файле по имени phone, в котором содержится список имен и номеров телефонов? Какой командой можно воспользоваться для вывода всего файла в алфавитном порядке? Как можно отобразить файл без смежных строк-дубликатов? Как можно отобразить файл без любых строк-дубликатов?
6. Что происходит, когда вы используете утилиту diff для сравнения содержимого двух двоичных, неидентичных файлов? (Для создания двоичных файлов можно воспользоваться утилитой gzip.) Объясните, почему данные, выводимые утилитой diff для двоичных файлов, отличаются от данных, выводимых этой утилитой для файлов формата ASCII.

7. Создайте файл .plan в своем личном каталоге. Отобразит ли утилита finger содержимое вашего файла .plan?
8. Что произойдет, если утилите which дать имя утилиты, которая находится в каталоге, не включенном в ваш путь поиска?
9. Существует ли утилита из числа рассмотренных в этой главе, находящаяся более чем в одном каталоге на локальной системе? Если да, то какая именно?
10. Проведите эксперимент, вызвав утилиту file с именами файлов в каталоге /usr/bin. Сколько различных типов файлов находится в этом каталоге?
11. Какой командой можно воспользоваться для просмотра первых нескольких строк файла по имени status.report? Какой командой можно воспользоваться для просмотра нескольких строк из конца этого файла?

Упражнения повышенной сложности

1. Создайте заново файлы colors.1 и colors.2, использовавшиеся в листинге 3.8. Проверьте свои файлы, запустив с именами этих файлов команду `diff -u`. Получатся ли у вас такие же результаты, как в этом листинге?
2. Попробуйте ввести эти две команды:

```
$ echo cat
$ cat echo
```

   Объясните разницу между информацией, выводимой каждой их них.
3. Повторите упражнение 5, используя файл phone.gz, сжатую версию списка имен и телефонных номеров. Рассмотрите более одного подхода для ответа на каждый вопрос и объясните, чем вы руководствовались в своем выборе.
4. Найдите существующие файлы или создайте файлы, которые
   а) утилита gzip сжимает более чем на 80 процентов;
   б) утилита gzip сжимает менее чем на 10 процентов;
   в) становятся больше по размеру при сжатии с помощью утилиты gzip.

   Используйте команду `ls -l` для определения размеров файлов, по которым заданы вопросы. Можете ли вы дать характеристику файлам, использованным в примерах а, б и в?
5. Старые программы электронной почты не могли работать с двоичными файлами. Предположим, что вы отправляете по электронной почте файл, сжатый утилитой gzip, которая производит двоичный файл, а получатель пользуется старой почтовой программой. Обратитесь к man-странице, посвященной утилите uuencode, которая выполняет преобразование двоичного файла в формат ASCII. Изучите характеристики утилиты и порядок ее использования.
   а) Превратите сжатый файл в файл формата ASCII, используя утилиту uuencode. Каким получился закодированный файл, больше или меньше сжатого? Обоснуйте ответ. (Если утилиты uuencode нет на локальной системе, ее можно установить, воспользовавшись утилитой yum или утилитой aptitude, которые используются для поддержания системы на современном уровне. Искомая утилита является частью пакета sharutils.)
   б) Есть ли вообще смысл применять утилиту uuencode к файлу перед его сжатием? Обоснуйте ответ.

Глава 4. Файловая система

Файловая система — это набор *данных*, который обычно находится на части диска и содержит каталоги с файлами. В файловой системе хранятся пользовательские и системные данные, составляющие основу для работы пользователя в системе и основу существования самой системы. В этой главе рассматривается файловая система Linux и используемая в ней терминология, даются определения обычным файлам и каталогам и объясняются правила, согласно которым им присваиваются имена. Демонстрируются также порядок создания и удаления каталогов, перемещения по файловой системе и использования абсолютных и относительных имен для доступа к файлам в различных каталогах. В главе также рассматриваются вопросы, касающиеся наиболее важных каталогов и файлов, а также прав доступа к файлам и списков управления доступом — Access Control Lists (ACL), которые позволяют организовать совместный (с другими пользователями) доступ к избранным файлам. В заключительной части главы рассматриваются жесткие и символьные ссылки, позволяющие одному и тому же файлу появляться более чем в одном каталоге.

Кроме этого, при чтении главы может возникнуть потребность обратиться к подробным описаниям утилит df, fsck, mkfs и tune2fs, приведенным в части V, чтобы получить дополнительные сведения о файловой системе.

Иерархическая файловая система

Генеалогическое древо

Иерархическая структура зачастую принимает форму пирамиды. Один из примеров такой структуры — изображение семейной родословной: у семейной пары есть ребенок, у которого, в свою очередь, может быть несколько детей, у каждого из которых тоже могут быть дети. Эта иерархическая структура называется *генеалогическим древом* (рис. 4.1).

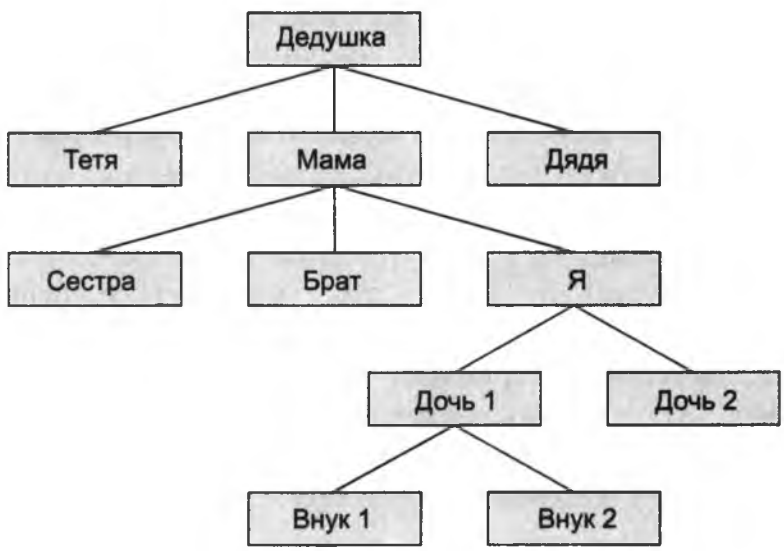

Рис. 4.1. Генеалогическое древо

Дерево каталогов

Благодаря сходству с генеалогическим древом файловая система Linux тоже называется деревом. Она состоит из набора связанных друг с другом файлов. Эта структура позволяет организовать файлы таким образом, чтобы можно было легко найти любой конкретный файл. На стандартной Linux-системе каждый пользователь начинает работу с одного каталога, к которому он может добавлять подкаталоги на любом желаемом уровне. Благодаря возможности выбора уровней каталогов пользователь может расширять структуру в соответствии со своими потребностями.

Подкаталоги

Обычно каждый каталог предназначен для информации, относящейся к отдельно взятой теме. Это может быть личная информация, проект или событие. Тема определяет, нуждается ли этот подкаталог в дальнейшем делении на подкаталоги. К примеру, на рис. 4.2 показан рабочий подкаталог секретаря, названный correspond (переписка). В этом каталоге содержатся три подкаталога: business (деловая), memos (служебная) и personal (личная).

В каталоге business содержатся файлы, хранящие каждое письмо, напечатанное секретарем. Если предполагается, что в адрес одного клиента, например в молочную компанию, будет отправлено множество писем, то для этого клиента можно выделить отдельный подкаталог milk_co.

Одно из достоинств файловой системы Linux заключается в ее способности подстраиваться под нужды пользователей. Стратегический подход к организации хранения своих файлов в наиболее полезной и удобной структуре дает массу преимуществ.

Файлы каталогов и обычные файлы

Как и генеалогическое древо, древовидная структура, представляющая файловую систему, обычно рисуется сверху вниз, с корневым каталогом, расположенным в самом верху. На рис. 4.2 и 4.3 показаны деревья, «растущие» из корня вниз, с путями, соединяющими корень с каждым из других файлов. В конце каждого пути находится либо обычный файл, либо файл каталога. Специальные файлы, которые также могут встречаться в конце путей, предоставляют доступ к функциям операционной системы. *Обычные файлы*, или просто *файлы*, встречаются в конце таких путей, которые не могут поддерживать другие пути. *Файлы каталогов*, которые также называют *каталогами* или *папками*, являются точками, от которых могут ответвляться другие пути. (На рис. 4.2 и 4.3 показано несколько пустых каталогов.)

При ссылках на древовидную структуру движение *вверх* означает движение по направлению к корню, а движение *вниз* — это движение по направлению от корня. Каталоги, непосредственно связанные путем, называются *родительскими* (те из них, которые расположены ближе к корневому каталогу) и *дочерними* (расположенные дальше от корневого каталога). *Полное имя* представляет собой последовательность имен, по которым прослеживается путь вдоль ветвей от одного файла к другому. Дополнительные сведения о полных именах можно найти в данной главе в разделе «Полные имена».

Дерево каталогов

Благодаря сходству с генеалогическим древом файловая система Linux тоже называется деревом. Она состоит из набора связанных друг с другом файлов. Эта структура позволяет организовать файлы таким образом, чтобы можно было легко найти любой конкретный файл. На стандартной Linux-системе каждый пользователь начинает работу с одного каталога, к которому он может добавлять подкаталоги на любом желаемом уровне. Благодаря возможности выбора уровней каталогов пользователь может расширять структуру в соответствии со своими потребностями.

Подкаталоги

Обычно каждый каталог предназначен для информации, относящейся к отдельно взятой теме. Это может быть личная информация, проект или событие. Тема определяет, нуждается ли этот подкаталог в дальнейшем делении на подкаталоги. К примеру, на рис. 4.2 показан рабочий подкаталог секретаря, названный correspond (переписка). В этом каталоге содержатся три подкаталога: business (деловая), memos (служебная) и personal (личная).

В каталоге business содержатся файлы, хранящие каждое письмо, напечатанное секретарем. Если предполагается, что в адрес одного клиента, например в молочную компанию, будет отправлено множество писем, то для этого клиента можно выделить отдельный подкаталог milk_co.

Одно из достоинств файловой системы Linux заключается в ее способности подстраиваться под нужды пользователей. Стратегический подход к организации хранения своих файлов в наиболее полезной и удобной структуре дает массу преимуществ.

Файлы каталогов и обычные файлы

Как и генеалогическое древо, древовидная структура, представляющая файловую систему, обычно рисуется сверху вниз, с корневым каталогом, расположенным в самом верху. На рис. 4.2 и 4.3 показаны деревья, «растущие» из корня вниз, с путями, соединяющими корень с каждым из других файлов. В конце каждого пути находится либо обычный файл, либо файл каталога. Специальные файлы, которые также могут встречаться в конце путей, предоставляют доступ к функциям операционной системы. *Обычные файлы*, или просто *файлы*, встречаются в конце таких путей, которые не могут поддерживать другие пути. *Файлы каталогов*, которые также называют *каталогами* или *папками*, являются точками, от которых могут ответвляться другие пути. (На рис. 4.2 и 4.3 показано несколько пустых каталогов.)

При ссылках на древовидную структуру движение *вверх* означает движение по направлению к корню, а движение *вниз* — это движение по направлению от корня. Каталоги, непосредственно связанные путем, называются *родительскими* (те из них, которые расположены ближе к корневому каталогу) и *дочерними* (расположенные дальше от корневого каталога). *Полное имя* представляет собой последовательность имен, по которым прослеживается путь вдоль ветвей от одного файла к другому. Дополнительные сведения о полных именах можно найти в данной главе в разделе «Полные имена».

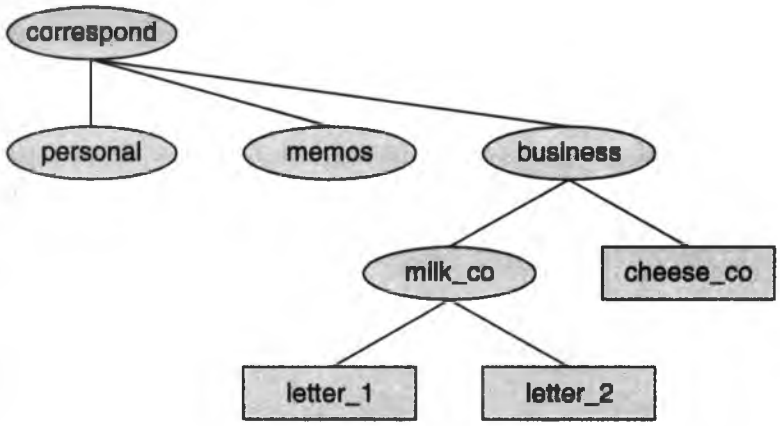

Рис. 4.2. Каталоги секретаря

Рис. 4.3. Файлы каталога и обычные файлы

Имена файлов

У каждого файла есть *имя*. Максимальная длина имени файла варьируется в зависимости от типа файловой системы. Linux поддерживает несколько типов файловых систем. Хотя многие современные файловые системы допускают использование файлов с именами длиной до 255 символов, некоторые файловые системы ограничивают имена файлов меньшим количеством символов. В именах файлов можно использовать почти все символы, но, во избежание хаоса, для них лучше использовать символы, перечисленные в следующем списке:

- Буквы верхнего регистра (A–Z).
- Буквы нижнего регистра (a–z).
- Цифры (0–9).
- Знак подчеркивания (_).
- Точка (.).
- Запятая (,).

Как и дети одного родителя, никакие два файла в одном и том же каталоге не могут иметь одно и то же имя. (Родители дают своим детям разные имена исходя из здравого смысла, а система Linux просто выдвигает такое требование.) Файлы в разных каталогах, подобно детям разных родителей, могут иметь одно и то же имя.

Выбираемые вами имена файлов должны быть осмысленными. Довольно часто каталог заполнен важными файлами с такими бесполезными именами, как hold1, wombat и junk, не говоря уже о таких именах, как foo и foobar. Подобные имена лучше не выбирать, поскольку они никак не помогут вам вспомнить, что именно хранится в конкретном файле. Следующие имена не только соответствуют рекомендованному синтаксису, но и содержат информацию о содержимом файла:

- correspond (переписка);
- january (январь);
- davis (дэвис);
- reports (отчеты);
- 2001;
- acct_payable (счета_к_оплате).

Длина имен файлов

При совместном использовании своих файлов с пользователями других систем может понадобиться, чтобы длинные имена файлов отличались друг от друга первыми несколькими символами. Системы, на которых запущена DOS или более ранние версии Windows, используют имена файлов, тело которых ограничено восемью, а расширение — тремя символами. У некоторых UNIX-систем имеется 14-символьное ограничение, а на старых системах Macintosh — 31-символьное ограничение. Если выбирать для файлов короткие имена, их будет проще набирать, а чуть позже к этим именам можно будет добавить расширения, не превышая при этом более жестких ограничений, накладываемых некоторыми файловыми системами. Недостаток коротких имен заключается в их меньшей информативности по сравнению с длинными.

Утилита stat (см. часть V) может вывести максимально допустимую длину имени файла. В следующем примере число, которое следует за строковым значением `Namelen:`, является максимальным количеством символов, допустимым в имени файла указанной файловой системы (`/home`).

```
$ stat -f /home | grep -i name
    ID: ff1f5275f648468b Namelen: 255     Type: ext2/ext3
```

Ключ `-f` заставляет утилиту stat вместо статуса файла вывести статус файловой системы. Ключ `-i` (case insensitive — отключить зависимость от регистра букв) заставляет утилиту grep при осуществлении поиска не обращать внимания на регистр букв.

Длинные имена файлов позволяют делать имена файлов более информативными. Чтобы помочь выбрать нужный файл без ввода полных имен файлов, оболочка предоставляет функцию автозавершения. Дополнительная информация об этом свойстве может быть найдена в совете «Автозавершение имени файла» на с. 69.

Чувствительность к регистру букв

В именах файлов можно использовать буквы в верхнем и (или) нижнем регистре, но при этом нужно соблюдать осторожность: многие файловые системы обладают чувствительностью к регистру букв. Например, популярное семейство файловых систем ext и файловая система UFS

обладают чувствительностью к регистру букв, поэтому имена JANUARY, January и january ссылаются на три разных файла. Семейство файловых систем FAT (используемое в основном для переносных носителей информации) не обладает чувствительностью к регистру букв, поэтому приведенные выше три имени представляют один и тот же файл. Файловая система HFS+, которая по умолчанию используется в OS X, относится к системам, сохраняющим регистр букв, но не обладающим чувствительностью к нему.

НЕ ИСПОЛЬЗУЙТЕ ПРОБЕЛЫ В ИМЕНАХ ФАЙЛОВ

Хотя использование пробелов в именах файлов не запрещено, от них все же лучше отказаться. Поскольку пробел относится к особым символам, в командной строке его нужно цитировать. Цитирование символа в командной строке может вызывать трудности у новичков и стать обременительной задачей для опытных пользователей. Вместо пробелов лучше использовать точки или знаки подчеркивания: joe.05.04.26, new_stuff.

Если приходится работать с именами файлов, содержащими пробелы, например с файлами из других операционных систем, нужно цитировать пробел в командной строке, устанавливая перед ним символ обратного слэша или устанавливая кавычки по обе стороны имени файла. Обе следующие команды отправляют файл по имени my file на печать.

```
$ lpr my\ file
$ lpr "my file"
```

Расширения имен файлов

Расширение имени файла является его частью, которая следует за встроенной в это имя точкой. В именах файлов, перечисленных в табл. 4.1, расширения имен помогают дать описание содержимому этих файлов. Некоторые программы, например компилятор языка программирования C, по умолчанию настроены на определенные расширения имен файлов, но в большинстве случаев расширения являются необязательным элементом. Расширения можно применять вполне свободно, чтобы сделать имена файлов более понятными. Если хотите, можете использовать внутри имени одного и того же файла сразу несколько точек, например notes.4.10.01 или files.tar.gz. При работе в операционной системе Mac OS X некоторые приложения используют расширения имен файлов, но многие приложения используют коды типа или коды создателя файла.

Таблица 4.1. Расширения имен файлов

Имя файла с расширением	Значение расширения
compute.c	Исходный файл на языке программирования C
compute.o	Объектный код для compute.c
compute	Исполняемый файл для compute.c
memo.0410.txt	Текстовый файл
memo.pdf	PDF-файл; в GUI просмотр его содержимого осуществляется с помощью утилиты xpdf или утилиты kpdf
memo.ps	PostScript-файл в GUI; просмотр его содержимого осуществляется с помощью утилиты gs или утилиты kpdf
memo.Z	Файл, сжатый утилитой compress; для возвращения его в обычный размер и формат используется утилита uncompress или утилита gunzip
memo.tgz или memo.tar.gz	tar-архив файлов, сжатый утилитой gzip

продолжение

Таблица 4.1 (*продолжение*)

Имя файла с расширением	Значение расширения
memo.gz	Файл, сжатый утилитой gzip; просмотр содержимого осуществляется с помощью утилиты zcat, а возвращение в обычный размер и формат — с помощью утилиты gunzip
memo.bz2	Файл, сжатый утилитой bzip2; просмотр содержимого осуществляется с помощью утилиты bzcat, а возвращение в обычный размер и формат — с помощью утилиты bunzip2
memo.html	Файл, предназначенный для просмотра в веб-браузере, например в Firefox
photo.gif, photo.jpg, photo.jpeg, photo.bmp, photo.tif, or photo.tiff	Файл, содержащий графическую информацию, например рисунок

Скрытые имена файлов

Имена файлов, начинающиеся с точки, называются *скрытыми именами файлов* (а файлы с такими именами называются *скрытыми* или *невидимыми файлами*), поскольку утилита ls обычно их не показывает. Команда `ls` с параметром `-a` выводит все имена файлов, даже скрытые. Имена файлов запуска (см. раздел «Файлы запуска» на с. 103) обычно начинаются с точки, поэтому они относятся к скрытым и не загромождают собой список содержимого каталога.

Файл .plan (см. главу 3, раздел «.plan и .project») также является скрытым. В каждом каталоге имеются еще и два специальных скрытых элемента — одинарная и двойная точка (. и ..) (см. раздел «Элементы каталога . и ..» на с. 108).

Рабочий каталог

Утилита pwd

При входе в систему Linux с использованием символьного интерфейса ваша работа всегда связана с каталогом, который называется *рабочим* или *текущим* каталогом. Иногда эта связь имеет под собой вполне определенный физический смысл: «Вы находитесь (или работаете) в каталоге zach». Встроенная в оболочку утилита pwd (print working directory — вывод рабочего каталога) (см. главу 5, раздел «Встроенные команды») выводит полное имя рабочего каталога.

Ваш личный каталог

При первом входе в систему в качестве рабочего выступает ваш *личный каталог*. Для отображения полного имени вашего личного каталога следует сразу же после входа в систему воспользоваться утилитой pwd (см. листинг 4.1). Личные каталоги в системе Linux обычно расположены в каталоге /home, а в Mac OS X они расположены в каталоге /Users.

При вызове без аргументов утилита ls выводит список файлов в рабочем каталоге. Поскольку ваш личный каталог до сих пор был единственным используемым рабочим каталогом, утилита ls всегда выводила список файлов, находящихся в этом каталоге. (Все файлы, созданные вами на данный момент, были созданы в вашем личном каталоге.)

Файлы запуска

Файлы запуска, появляющиеся в вашем личном каталоге, снабжают оболочку и другие программы информацией о вас и ваших предпочтениях. При работе в Mac OS X эти файлы называются *конфигурационными файлами* или *файлами предпочтений*. Зачастую один из этих файлов сообщает оболочке, какой тип терминала вы используете, и запускает утилиту stty (set terminal — настроить терминал) для установки клавиш стирания символа (см. раздел «Удаление символа» на с. 48) и удаления строки (см. раздел «Удаление строки» на с. 49).

Листинг 4.1. Вход в систему и вывод путевого имени вашего личного каталога

```
login: max
Password:
Last login: Wed Oct 20 11:14:21 from bravo
$ pwd
/home/max
```

Поместить в ваш личный каталог файлы запуска оболочки, содержащие ее команды, можете либо вы сами, либо системный администратор. Оболочка выполняет команды, находящиеся в этом файле, при каждом вашем входе в систему. Поскольку у файлов запуска скрытые имена, чтобы убедиться в наличии такого файла в вашем личном каталоге, нужно воспользоваться командой `ls -a`. Дополнительные сведения о файлах запуска можно найти в разделе «Файлы запуска» на с. 284 (для оболочки bash) и в одноименном разделе в главе 9 (для оболочки tcsh).

Полные имена

У каждого файла есть *полное имя*, которое отслеживается по каталогам при проходе через часть их иерархии к обычному файлу или каталогу. В полном имени символ слэша (/) справа от имени файла свидетельствует о том, что этот файл является файлом каталога. Файл, имя которого следует за символом слэша, может быть обычным файлом или файлом каталога. Простейшим полным именем является само имя файла, указывающее на файл в рабочем каталоге. В этом разделе рассматриваются абсолютные и относительные полные имена и даются объяснения, как пользоваться каждым из них.

Абсолютные полные имена

/ (корневой каталог)

У корневого каталога в иерархии файловой системы собственное имя отсутствует. Ссылка на него осуществляется именно как на *корневой каталог* и представлена символом слэша (/), как отдельно стоящего, так и находящегося в крайней левой позиции имени.

Абсолютное имя начинается с символа слэша (/), который представляет собой корневой каталог. За этим символом следует имя файла, расположенного в корневом каталоге. Абсолютное имя продолжается по маршруту пути, ведущему через все промежуточные каталоги к файлу, определяемому полным именем. Все имена файлов на этом пути составляются в одну строку, и после каждого каталога ставится символ слэша (/). Эта строка имен файлов называется абсолютным именем, поскольку она

определяет местонахождение файла абсолютно, отслеживая путь от корневого каталога до данного файла. Обычно абсолютный путь к каталогу не включает в себя замыкающий слэш, хотя этот формат может быть использован, чтобы подчеркнуть, что путевое имя указывает на каталог (например, /home/zach/). Та часть полного имени, которая следует за последним символом слэша, называется *простым именем файла*, *именем файла* или *базовым именем*. На рис. 4.4 показаны абсолютные имена каталогов и обычных файлов применительно к части иерархии файловой системы.

Используя абсолютное имя (если вам предоставлены соответствующие права), можно вывести данные или работать каким-нибудь другим образом с любым файлом на локальной системе, независимо от того, в каком рабочем каталоге вы находитесь на момент ввода команды. Например, пользователь Sam, работая в своем личном каталоге, может ввести команду вывода списка файлов, которые находятся в каталоге /usr/bin:

```
$ pwd
/home/sam
$ ls /usr/bin
7z                              kwin
7za                             kwin_killer_helper
822-date                        kwin_rules_dialog
...
```

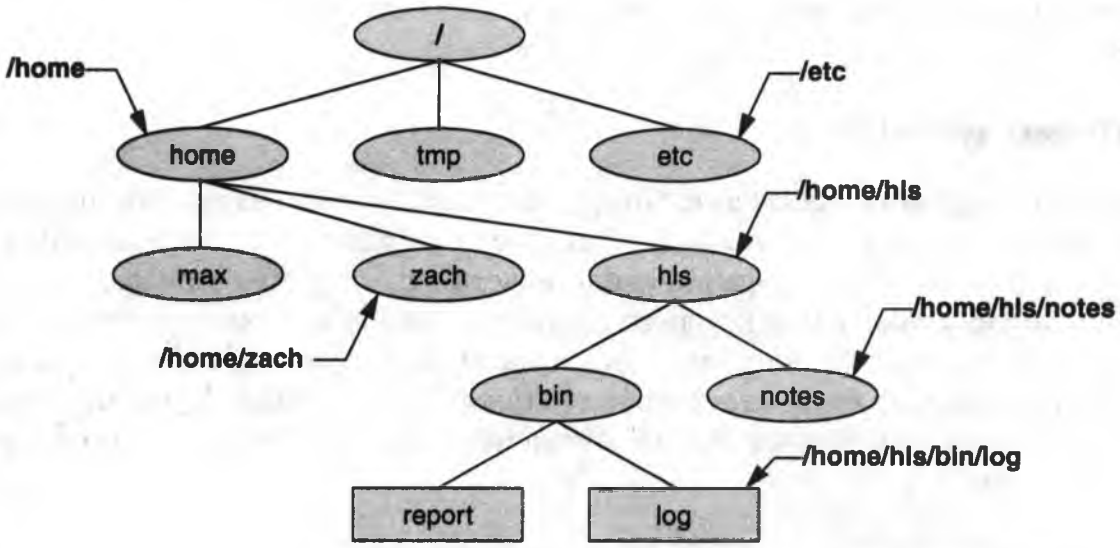

Рис. 4.4. Абсолютные имена

Знак ~ (тильда) в полных именах

При использовании еще одной формы абсолютного имени оболочка превращает сочетание символов ~/ (тильда, за которой следует слэш), которое стоит в начале имени, в полное имя вашего личного каталога. Используя это сокращение, можно вывести содержимое вашего файла запуска .bashrc (см. раздел «Настройка файлов запуска» на с. 286) с помощью такой команды (независимо от того, какой из каталогов является на данный момент рабочим):

```
$ less ~/.bashrc
```

Тильда позволяет быстро ссылаться на пути, которые начинаются в вашем или в чьем-нибудь личном каталоге. Оболочка расширяет тильду, за которой в самом начале полного имени следует имя пользователя, в полное имя личного каталога этого пользователя. Например (предполагая, что он имеет на это право), пользователь Max может просмотреть файл .bashrc, принадлежащий пользователю Sam, с помощью следующей команды:

```
$ less ~sam/.bashrc
```

Дополнительные сведения можно найти в разделе «Расширение тильды» на с. 355.

Относительные имена

В *относительных именах* отслеживается путь от рабочего каталога к файлу. Любое имя, которое не начинается с корневого каталога (представляемого символом /) или тильды (~), является относительным. В относительных именах, как и в абсолютных, могут отслеживаться пути, проходящие через множество каталогов. Простейшим относительным именем является простое имя файла, которое идентифицирует файл в рабочем каталоге. Примеры в следующем разделе используют абсолютные и относительные имена.

Значение рабочего каталога

Для доступа к любому файлу в рабочем каталоге нужно указать только простое имя файла. Для доступа к файлу из другого каталога нужно использовать полное имя. Набирать абсолютное имя — слишком утомительное занятие, к тому же повышающее ваши шансы на ошибку. Такой набор менее характерен для работы в графическом интерфейсе (GUI), где используется щелчок на именах файлов или значках. Чтобы уменьшить потребности в длинных именах, можно выбрать рабочий каталог для любой конкретной задачи. Выбор рабочего каталога не дает каких-то особых преимуществ, позволяющих сделать что-нибудь такое, чего нельзя было бы сделать каким-нибудь другим образом — он всего лишь упрощает некоторые операции.

ПРИ ИСПОЛЬЗОВАНИИ ОТНОСИТЕЛЬНОГО ИМЕНИ НУЖНО ЗНАТЬ, КАКОЙ ИЗ КАТАЛОГОВ ЯВЛЯЕТСЯ РАБОЧИМ

Местонахождение файла, к которому осуществляется доступ с помощью относительного имени, зависит от рабочего каталога (или определяется по отношению к нему). Перед использованием относительного имени нужно всегда быть уверенным, что вы знаете, какой именно каталог является рабочим. Для проверки каталога нужно воспользоваться утилитой pwd. Если вы создаете файл, используя редактор vim, и ошиблись в своем местонахождении в файловой иерархии, новый файл может оказаться в совершенно неожиданном для вас месте.

Когда используется абсолютное имя, понятие рабочего каталога не играет никакой роли.

Таким образом, следующая команда всегда приводит к редактированию файла по имени goals, который находится в вашем личном каталоге:

```
$ vim ~/goals
```

При чтении этого абзаца следует обратить внимание на рис. 4.5. На файлы, дочерние по отношению к рабочему каталогу, можно ссылаться с помощью простых имен файлов. На файлы, которые можно считать «внуками» рабочего каталога, можно ссылаться с помощью коротких относительных имен: двух имен файлов, разделенных

символом слэша. При работе с файлами в развитой структуре каталогов использование коротких относительных имен поможет сэкономить время и нервы. Если в качестве рабочего выбрать тот каталог, в котором находятся файлы, наиболее часто используемые при решении конкретной задачи, можно будет использовать менее длинные и громоздкие имена. (Об элементах . и .. см. раздел «Элементы каталога . и ..» на с. 108.)

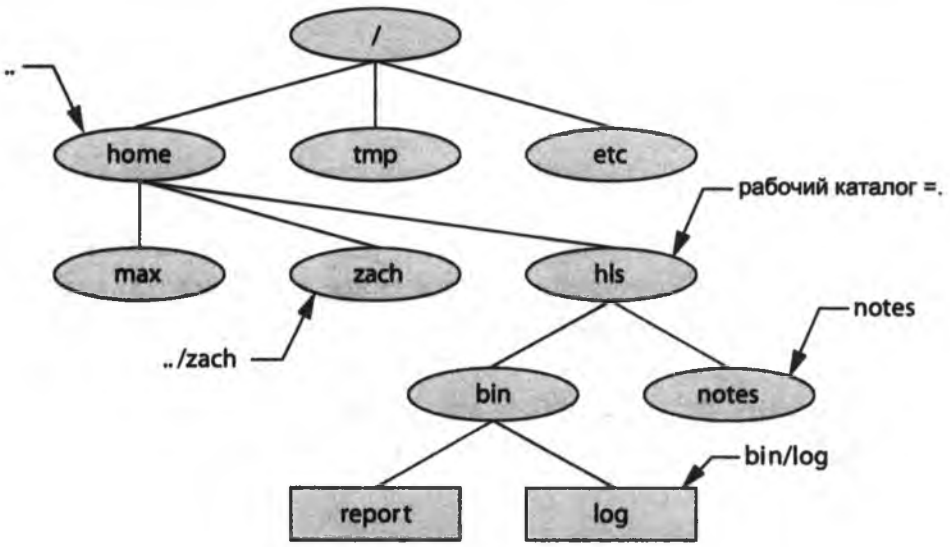

Рис. 4.5. Относительные имена

Работа с каталогами

В этом разделе рассматривается создание каталогов (утилита mkdir), переходы по каталогам (утилита cd), удаление каталогов (утилита rmdir), использование абсолютных и относительных имен для упрощения своей работы, перемещение и копирование файлов из одних каталогов в другие. В заключительной части приводится список и даются краткие описания важных стандартных каталогов и файлов, принадлежащих файловой системе Linux.

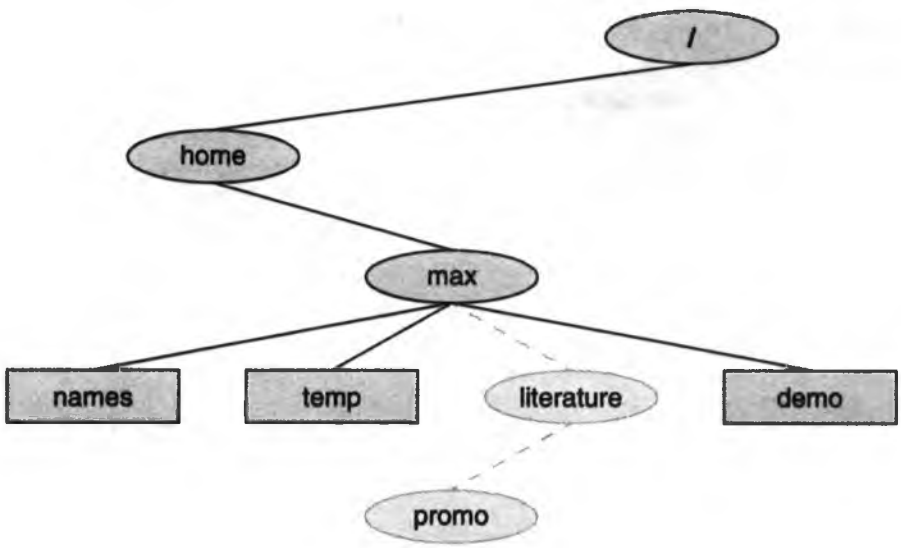

Рис. 4.6. Файловая структура, разрабатываемая в примерах

mkdir: Создание каталога

Утилита mkdir позволяет создать каталог. Аргумент утилиты mkdir становится полным именем нового каталога. В следующих примерах разрабатывается структура каталогов, показанная на рис. 4.6. На этом рисунке добавляемые каталоги изображены немного светлее всех остальных и соединены пунктирными линиями.

В листинге 4.2 утилита pwd показывает, что пользователь Max работает в своем личном каталоге (/home/max), а утилита ls показывает имена файлов в его личном каталоге: demo, names и temp.

Используя утилиту mkdir, Max создает каталог по имени literature в качестве дочернего по отношению к его личному каталогу. Он использует относительное имя (простое имя файла), поскольку хочет, чтобы каталог literature был дочерним по отношению к рабочему каталогу. Пользователь Max мог бы воспользоваться для создания такого же каталога абсолютным именем: `mkdir /home/max/literature` или вариантом `mkdir ~max/literature`.

Вторая команда ls, показанная в листинге 4.2, позволяет проверить существование нового каталога. Ключ `-F` при вызове утилиты ls предназначен для отображения символа слэша после имени каждого каталога и звездочки после имени каждого исполняемого файла (сценария оболочки, утилиты или приложения). Если при вызове утилиты ls в качестве аргумента использовать имя каталога, она выводит список содержимого этого каталога. Последняя команда ls не выводит ничего, поскольку в каталоге literature нет файлов.

Листинг 4.2. Утилита mkdir

```
$ pwd
/home/max
$ ls
demo  names  temp
$ mkdir literature
$ ls
demo  literature  names  temp
$ ls -F
demo  literature/  names  temp
$ ls literature
$
```

Следующие команды показывают два способа создания каталога promo, являющегося дочерним по отношению к только что созданному каталогу literature. В первом из них проверяется, что /home/max является рабочим каталогом, и используется относительное имя:

```
$ pwd
/home/max
$ mkdir literature/promo
```

Во втором используется абсолютное имя:

```
$ mkdir /home/max/literature/promo
```

Для создания одной командой сразу двух каталогов, literature и promo, нужно при вызове утилиты mkdir воспользоваться ключом `-p` (parents — родители):

```
$ pwd
/home/max
```

```
$ ls
demo  names  temp
$ mkdir -p literature/promo
```

или

```
$ mkdir -p /home/max/literature/promo
```

cd: Смена рабочего каталога

Утилита cd (change directory — сменить каталог) делает рабочим другой каталог, но не изменяет содержимое рабочего каталога. В листинге 4.3 показаны два способа смены рабочего каталога на /home/max/literature, что проверяется с помощью утилиты pwd. Сначала пользователь Max использует утилиту cd с абсолютным именем, чтобы сделать literature своим рабочим каталогом. При вводе команды с абсолютным именем совершенно неважно, какой каталог является рабочим.

Команда pwd подтверждает изменения, сделанные пользователем Max. При использовании без аргументов утилита cd превращает в рабочий ваш личный каталог, как это было при входе в систему. Вторая команда cd, показанная в листинге 4.3, не имеет аргументов, поэтому она превращает личный каталог пользователя Max в рабочий каталог. И наконец, зная о том, что он работает в своем личном каталоге, Max использует простое имя файла для превращения каталога literature в свой рабочий (cd literature) и убеждается в успехе задуманного с помощью команды pwd.

Листинг 4.3. Утилита cd производит смену рабочего каталога

```
$ cd /home/max/literature
$ pwd
/home/max/literature
$ cd
$ pwd
/home/max
$ cd literature
$ pwd
/home/max/literature
```

СРАВНЕНИЕ РАБОЧЕГО И ЛИЧНОГО КАТАЛОГОВ

Рабочий каталог не нужно путать с личным. Личный каталог остается одним и тем же на протяжении всего сеанса работы, а также, как правило, от сеанса к сеансу. Сразу же после входа в систему вы всегда работаете в одном и том же каталоге, в качестве которого выступает ваш личный каталог.

В отличие от личного каталога рабочий каталог можно менять так часто, как вы этого захотите. Рабочий каталог не нужно устанавливать, чем объясняется тот факт, что некоторые люди предпочитают называть его текущим каталогом. После входа в систему и до смены каталогов с помощью утилиты cd ваш личный каталог является рабочим. Если бы вы сменили каталог на личный каталог пользователя Sam, рабочим стал бы именно этот каталог.

Элементы каталога . и ..

Утилита mkdir автоматически помещает в каждый создаваемый каталог два элемента: одинарную (.) и двойную (..) точку. Посмотрите на рис. 4.5. Точка (.) является синонимом имени рабочего каталога и может быть использована вместо него, а двойная точка (..) является синонимом имени того каталога, который является родительским

по отношению к рабочему каталогу. Эти элементы относятся к скрытым, потому что имена их файлов начинаются с точки.

В следующем примере при рабочем каталоге literature двойная точка (..) используется трижды: в первый раз для вывода списка содержимого родительского каталога (/home/max), во второй — для копирования файла memoA в родительский каталог и в третий — опять для вывода списка содержимого родительского каталога.

```
$ pwd
/home/max/literature
$ ls ..
demo  literature  names  temp
$ cp memoA ..
$ ls ..
demo  literature  memoA  names  temp
```

После вызова утилиты cd для превращения каталога promo (который по отношению к каталогу literature является его подкаталогом) в свой рабочий каталог Max может при вызове программы vim для редактирования файла в своем личном каталоге использовать относительное имя.

```
$ cd promo
$ vim ../../names
```

Абсолютное или относительное имя, или простое имя файла, можно использовать практически везде, где утилита или программа требует имени файла или полного имени. Это справедливо для ls, vim, mkdir, rm и большинства других Linux-утилит.

rmdir: Удаление каталога

Удаление каталога осуществляется с помощью утилиты rmdir (remove directory — удалить каталог). Удалить рабочий каталог или каталог, в котором содержится что-нибудь, кроме элементов . и .., невозможно. Если нужно удалить каталог, в котором содержатся файлы, сначала нужно воспользоваться утилитой rm для удаления этих файлов, а затем уже удалить сам каталог. Вы не должны (да и не сможете) удалять элементы . и .., утилита rmdir удаляет их автоматически. Следующая команда удаляет каталог promo:

```
$ rmdir /home/max/literature/promo
```

С утилитой rm можно применять ключ -r (rm -r filename), который рекурсивно удаляет файлы и каталоги внутри указанного каталога, а также сам этот каталог.

ПРИ ИСПОЛЬЗОВАНИИ RM –R СЛЕДУЕТ ПРОЯВЛЯТЬ ОСОБУЮ ОСТОРОЖНОСТЬ

При всех удобствах, предоставляемых командой rm –r, ее нужно использовать крайне осмотрительно. Не используйте ее с нестрогими ссылками на файлы, например с символом звездочки (*). Эта команда пугает той простотой, с которой можно вводом одной короткой строки удалить весь свой личный каталог.

Использование путевых имен

Утилита touch Если хотите проверить в действии следующие примеры, создайте с помощью текстового редактора файл по имени letter. Для создания пустого файла можно также воспользоваться утилитой touch (см. часть V):

```
$ cd
$ pwd
/home/max
$ touch letter
```

В следующих примерах рабочим является каталог /home/max, и в нем используется утилита cp с относительным именем для копирования файла letter в каталог /home/max/literature/promo. (Нужно будет заново создать каталог promo, если ранее он уже был удален.) Копия файла имеет простое имя — letter.0610:

```
$ cp letter literature/promo/letter.0610
```

Если пользователь Max не меняет рабочий каталог, мы можем использовать программу vim для редактирования только что созданной копии:

```
$ vim literature/promo/letter.0610
```

Если Max не хочет использовать для указания нужного файла длинное полное имя, он может воспользоваться утилитой cd, чтобы перед вызовом vim превратить promo в свой рабочий каталог:

```
$ cd literature/promo
$ pwd
/home/max/literature/promo
$ vim letter.0610
```

Чтобы сделать новым рабочим каталогом родительский каталог (то есть каталог с именем /home/max/literature), Max может ввести следующую команду, в которой используется элемент каталога ..:

```
$ cd ..
$ pwd
/home/max/literature
```

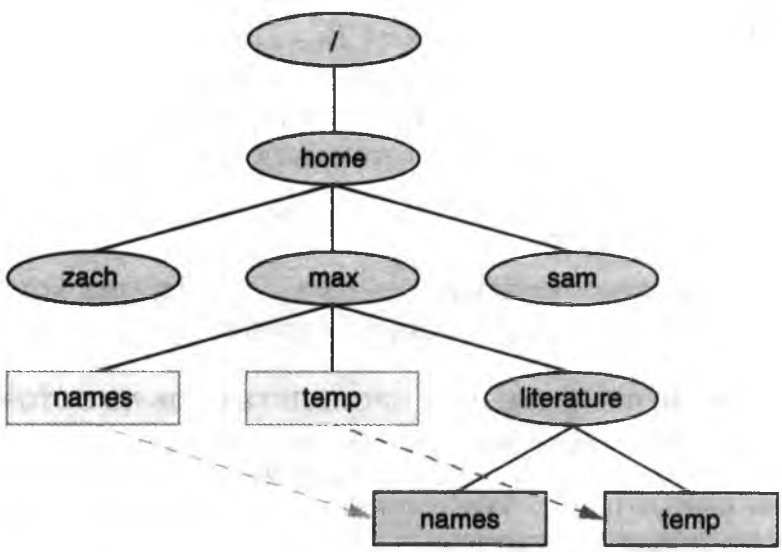

Рис. 4.7. Использование утилиты mv для перемещения файлов names и temp

mv, cp: Перемещение или копирование файлов

В главе 3 рассматривалось использование утилиты mv для переименования файлов. Но утилита mv обладает более универсальными возможностями: ее можно использовать

для перемещения файлов из одного каталога в другой (для изменения путевого имени файла), а также для изменения простого имени файла. Когда команда mv используется для перемещения одного или нескольких файлов в новый каталог, применяется следующий синтаксис:

```
mv список_существующих_файлов каталог
```

Если рабочим является каталог /home/max, пользователь Max для перемещения файлов names и temp из рабочего каталога в каталог literature может воспользоваться следующей командой:

```
$ mv names temp literature
```

Эта команда изменяет абсолютные путевые имена файлов names и temp с /home/max/names и с /home/max/temp на /home/max/literature/names и на /home/max/literature/temp соответственно (рис. 4.7). Как и многие другие команды Linux, mv воспринимает как абсолютные, так и относительные имена.

По мере работы в Linux и создания новых файлов, чтобы упорядочить их организацию, возникает потребность в создании новых каталогов с помощью утилиты mkdir. По мере расширения иерархии каталогов можно будет воспользоваться утилитой mv для перемещения файлов из одного каталога в другой.

Утилита cp работает так же, как и утилита mv, за исключением того, что она создает копии *списка_существующих_файлов* в указанном *каталоге*.

mv: Перемещение каталога

Утилита mv перемещает не только файлы, она точно так же перемещает из одного каталога в другой и целые каталоги. При этом используется похожий синтаксис, за исключением того, что вместо обычных файлов для перемещения указывается один или несколько каталогов:

```
mv список_существующих_каталогов новый_каталог
```

Если *новый_каталог* не существует, *список_существующих_каталогов* должен содержать только одно имя каталога, которое утилита mv изменит на *новый_каталог* (то есть mv изменяет имя каталога).

Несмотря на то что вы можете переименовать каталоги с помощью утилиты mv, скопировать их содержимое с помощью утилиты cp вы не можете, пока не воспользуетесь ключом -r (recursive — рекурсивно). Другие способы копирования и перемещения можно найти в части V, в описаниях утилит tar и cpio.

Важные стандартные каталоги и файлы

Изначально файлы в системе Linux не находились внутри иерархии каталогов в стандартных местах. Разбросанные по разным местам файлы затрудняли документирование и обслуживание системы Linux, что практически не позволяло выпускать пакеты программ, которые можно было бы компилировать и запускать на всех Linux-системах. Первый стандарт для файловой системы Linux, FSSTND (Linux Filesystem Standard) был выпущен в начале 1994 года. А в начале 1995 года началась работа над расширенным стандартом, распространяющимся на многие UNIX-подобные системы — FHS (Linux Filesystem Hierarchy Standard — стандарт файловой иерархии системы Linux;

proton.pathname.com/fhs). Позже FHS был включен в LSB (Linux Standard Base – основы стандарта Linux; www.linuxfoundation.org/en/LSB), рабочей группы FSG (Free Standards Group — группы свободных стандартов). И наконец, FSG объединилась с лабораторией разработки программ с открытым кодом — Open Source Development Labs (OSDL), чтобы сформировать Linux Foundation (www.linuxfoundation.org). На рис. 4.8 показано размещение некоторых важных каталогов и файлов в соответствии со спецификацией FHS. Значение многих из этих каталогов будет раскрыто в процессе изучения данной главы.

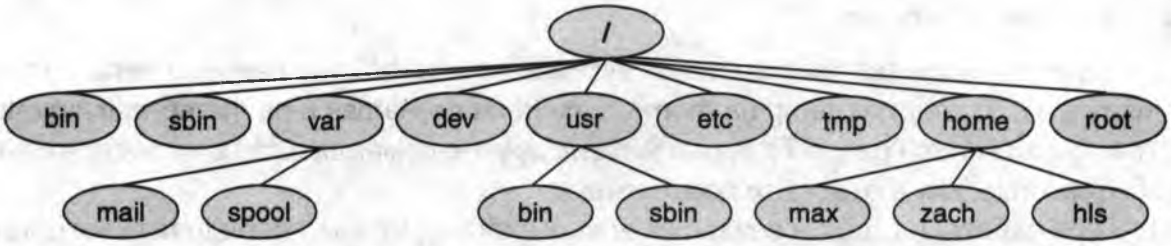

Рис. 4.8. Типичная структура файловой системы Linux, основанной на FHS

В следующем списке дано описание каталогов, показанных на рис. 4.8, часть из которых определена стандартом FHS. Большинство распространителей Linux не использует всех каталогов, определенных стандартом FHS. Нужно понимать, что определить функцию каталога по его имени удается не всегда. Например, в каталоге /opt (что означает необязательный) хранятся дополнительные программы, а в каталоге /etc/opt хранятся конфигурационные файлы для программ, находящихся в каталоге /opt.

/ **Корневой каталог.** Он присутствует во всех структурах файловых систем Linux и от него берут начало все файлы в файловой системе.

/bin **Основные двоичные командные файлы.** Содержит файлы, необходимые для загрузки системы и ее запуска, когда система сначала попадает в однопользовательский или восстановительный режим.

/boot **Статические файлы загрузчика.** Содержит все файлы, необходимые для загрузки системы.

/dev **Файлы устройств.** Содержит все файлы, представляющие внешние устройства, такие как приводы дисков, терминалы и принтеры. Ранее этот каталог был заполнен файлами всевозможных устройств. Утилита udev предоставляет динамический каталог устройства, что позволяет каталогу /dev содержать только файлы устройств, присутствующих в системе.

/etc **Конфигурационные файлы для локальной машины.** Содержит администраторские, конфигурационные и другие системные файлы. Один из наиболее важных файлов — /etc/passwd, который содержит список всех пользователей, имеющих право пользования системой. В Mac OS X вместо /etc/passwd используется Open Directory.

/etc/opt **Конфигурационные файлы для пакетов дополнительного программного обеспечения, хранящихся в каталоге /opt.**

/etc/X11 **Конфигурационные файлы для системы X Window, работающей на локальной машине.**

/home **Личные каталоги пользователей.** Каждый личный каталог пользователя является, как правило, одним из множества подкаталогов каталога /home. Например, если предположить наличие пользовательских каталогов, входящих в состав каталога /home, то абсолютное путевое имя личного каталога пользователя Zach будет /home/zach. На некоторых системах личные каталоги могут встречаться не в каталоге /home, а быть разбросанными по другим каталогам, например /inhouse и /clients. В Mac OS X личные каталоги пользователей обычно находятся в каталоге /Users.

/lib **Общие библиотеки.**

/lib/modules **Загружаемые модули ядра.**

/mnt **Точка подключения (монтирования) временно подключаемых файловых систем.**

/opt **Пакеты дополнительного (необязательного) программного обеспечения.**

/proc **Виртуальная файловая система, содержащая информацию для ядра и процессов.**

/root **Личный каталог для учетной записи root.**

/sbin **Основные системные двоичные файлы.** Утилиты, используемые для системного администрирования хранятся в каталогах /sbin и /usr/sbin. В каталоге /sbin содержатся утилиты, необходимые для процесса загрузки системы, а в каталоге /usr/sbin содержатся утилиты, используемые после загрузки и запуска. На старых версиях Linux многие утилиты системного администрирования были разбросаны по нескольким каталогам, которые часто включали в себя другие системные файлы (/etc, /usr/bin, /usr/adm, /usr/include).

/sys **Псевдофайловая система устройств.**

/tmp **Временные файлы.**

/Users **Личные каталоги пользователей в Mac OS X.** Каждый личный каталог пользователя является, как правило, одним из множества подкаталогов каталога /Users. Linux обычно хранит личные каталоги пользователей в каталоге /home.

/usr **Вторая главная иерархия.** Традиционно включает подкаталоги, в которых содержится информация, используемая системой. Файлы в подкаталогах /usr не подвергаются частым изменениям и могут совместно использоваться несколькими системами.

/usr/bin **Основная часть пользовательских команд.** Содержит стандартные Linux/OS X утилиты, то есть двоичные файлы, которые не нужны в однопользовательском или восстановительном режиме.

/usr/games **Игры и обучающие программы.**

/usr/include **Заголовочные файлы, используемые программами на языке C.**

/usr/lib **Библиотеки.**

/usr/local **Локальная иерархия.** Содержит важные файлы и каталоги, добавляемые к системе. В эту иерархию могут включаться подкаталоги bin, games, include, lib, sbin, share и src.

/usr/sbin **Двоичные файлы системного администрирования, не имеющие жизненно важного значения.** См. каталог /sbin.

/usr/share **Архитектурно-независимые данные.** В состав могут входить подкаталоги dict, doc, games, info, locale, man, misc, terminfo и zoneinfo.

/usr/share/doc **Документация.**

/usr/share/info **Первичный каталог информационных систем GNU.**

/usr/share/man **Интерактивные руководства.**

/usr/src **Исходный код.**

/var **Переменные данные.** Файлы, содержимое которых изменяется по мере работы системы, и хранящиеся в подкаталогах, подчиненных каталогу /var. Самыми распространенными примерами могут послужить временные файлы, регистрационные файлы системы, файлы спулинга и файлы почтового ящика пользователя. В качестве подкаталогов могут быть каталоги cache, lib, lock, log, mail, opt, run, spool, tmp и yp. В старых версиях Linux подобные файлы были разбросаны по нескольким подкаталогам, подчиненным каталогу /usr (/usr/adm, /usr/mail, /usr/spool, /usr/tmp).

/var/log **Регистрационные файлы.** Содержат, кроме всего прочего, файл lastlog (запись последнего входа в систему каждого пользователя), файл messages (системные сообщения из файла syslogd) и файл wtmp (запись всех входов в систему и выходов из нее).

/var/spool **Буферизированные данные приложений (спулинг).** Содержат anacron, at, cron, lpd, mail, mqueue, samba и другие каталоги. Файл /var/spool/mail обычно ссылается на /var/mail.

Права доступа

В Linux поддерживаются два способа управления доступом пользователей к файлам и порядком их доступа: традиционные права доступа Linux и списки управления доступом — Access Control List (ACL). В этом разделе рассматриваются традиционные права доступа Linux. Списки управления доступом, предоставляющие более тонкую организацию прав доступа по сравнению с традиционными правами, рассмотрены ниже.

Получить доступ к файлу могут три типа пользователей: владелец файла (*владелец*), участник группы, с которой связан файл (*группа*), и любой другой пользователь (*все остальные*). Пользователь может попытаться получить доступ к обычному файлу тремя способами: попытаться *прочитать из него данные*, попытаться *записать в него данные* или попытаться *запустить его на выполнение*.

ls –l: Отображение прав

При вызове утилиты ls с ключом -l и с именем одного или нескольких обычных файлов она выводит строку с информацией о файле. В следующем примере выводится информация для двух файлов. Файл letter.0610 содержит текст письма, а файл check_spell содержит сценарий оболочки, программу, написанную на высокоуровневом языке программирования оболочки:

```
$ ls -l letter.0610 check_spell
-rwxr-xr-x 1 max pubs  852 Jul 31 13:47 check_spell
-rw-r--r-- 1 max pubs 3355 Jun 22 12:44 letter.0610
```

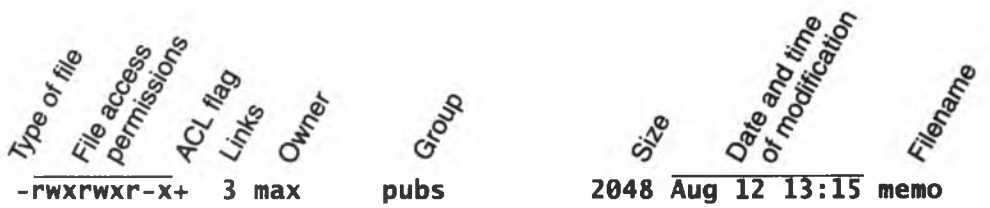

Рис. 4.9. Столбцы, отображаемые командой ls –l

Слева направо строка, выводимая командой `ls -l`, содержит следующую информацию (рис. 4.9):

- Тип файла (первый символ).
- Права доступа к файлу (следующие девять символов).
- Флаг ACL (присутствует, если у файла есть список управления доступом (см. раздел «Списки управления доступом (ACL-списки)» на с. 121).
- Количество ссылок на файл (см. раздел «Ссылки» на с. 126).
- Имя владельца файла (обычно это человек, создавший файл).
- Имя группы, с которой связан этот файл.
- Размер файла в символах (байтах).
- Дата и время создания или последней модификации файла.
- Имя файла.

Тип файла

Тип файла (первый столбец) для файла letter.0610 изображается дефисом (-), потому что это обычный файл. Файлы каталога в этом столбце содержат символ d. Список других символов, которые могут появляться в этой позиции, приведен в части V на рис. V.19.

Права доступа

Следующие три символа показывают права доступа для *владельца* файла: r (в первой позиции) показывает право на чтение, w (во второй позиции) показывает право на запись и x (в третьей позиции) показывает право на выполнение. Символ прочерка, или дефиса (-), в одной из позиций, показывает, что у владельца нет права доступа по этой позиции.

Точно так же следующие три символа представляют собой права доступа для *группы*, а последние три символа — для *всех остальных*. В предыдущем примере владелец файла letter.0610 может читать данные из этого файла и осуществлять запись данных в файл, а группа и все остальные могут только читать данные из файла, и никто из

всех этих категорий не обладает правами запуска файла на выполнение. Хотя права запуска на выполнение могут быть даны для любого файла, присваивать это право для файла, в котором содержится документ, например письмо, не имеет никакого смысла. Файл check_spell является выполняемым сценарием оболочки, поэтому право запуска на выполнение к нему вполне подходит. (Право запуска на выполнение имеют владелец, группа и все остальные.) Дополнительные сведения можно найти в подразделе «Комментарии» раздела, посвященного утилите ls на с. 757.

chmod: Изменение прав доступа

Используемая в Linux система прав доступа к файлу позволяет вам предоставить другим пользователям доступ к тем файлам, которыми вы с ними хотите поделиться, при этом сохранив конфиденциальность личных (закрытых) файлов. Можно разрешить другим пользователям считывать данные из файла и записывать их в файл (это удобно в том случае, когда вы участвуете в общем проекте, над которым работают сразу несколько людей). Можно разрешить всем остальным пользователям только считывать данные из файла (возможно, что этот файл — предлагаемая вами спецификация проекта). Или можно позволить всем остальным только записывать данные в файл (что похоже на ящик входящих сообщений или почтовый ящик, когда нужно, чтобы другие пользователи могли отправлять вам почту, но не могли ее читать). Аналогичным образом можно защитить целые каталоги от сканирования (к чему мы еще вернемся).

ПОЛЬЗОВАТЕЛЬ С ROOT-ПРАВАМИ МОЖЕТ ПОЛУЧИТЬ ДОСТУП К ЛЮБОМУ ФАЙЛУ СИСТЕМЫ

У прав доступа, рассматриваемых в данном разделе, есть одно исключение. Любой пользователь, которому могут быть предоставлены root-права, может получить и полный доступ ко всем файлам, независимо от того, кто их владелец или какие права доступа в них установлены.

Владелец файла управляет предоставлением прав доступа к нему для других пользователей, а также порядком этого доступа. Являясь владельцем файла, вы можете воспользоваться утилитой chmod (change mode — сменить режим) для изменения прав доступа к этому файлу. Утилите chmod можно указать символьные (относительные) или числовые (абсолютные) аргументы.

Символьные аргументы для утилиты chmod

В следующем примере применяются символьные аргументы для утилиты chmod, добавляющие (+) права на чтение и запись (rw) для всех (a) пользователей:

```
$ ls -l letter.0610
-rw------- 1 max pubs 3355 Jun 22 12:44 letter.0610
$ chmod a+rw letter.0610
$ ls -l letter.0610
-rw-rw-rw- 1 max pubs 3355 Jun 22 12:44 letter.0610
```

Использование с утилитой chmod символьных аргументов приводит к изменению прав доступа. Изменения, производимые заданным аргументом, зависят от (задаются относительно) существующих прав доступа. В следующем примере утилита chmod удаляет (-) права на чтение (r) и выполнение (x) для всех остальных (o) пользователей. Прав владельца и участников группы она не касается.

```
$ ls -l check_spell
-rwxr-xr-x 1 max pubs 852 Jul 31 13:47 check_spell
$ chmod o-rx check_spell
$ ls -l check_spell
-rwxr-x--- 1 max pubs 852 Jul 31 13:47 check_spell
```

ДЛЯ ЗАПУСКА СЦЕНАРИЯ ОБОЛОЧКИ У ВАС ДОЛЖНО БЫТЬ ПРАВО НА ЧТЕНИЕ ФАЙЛА

Поскольку оболочке перед выполнением команд, содержащихся в сценарии, нужно прочитать свой сценарий (текстовый файл, содержащий команды оболочки), перед выполнением команд, которые находятся в сценарии, вы должны иметь права на чтение того файла, который содержит сценарий, чтобы иметь возможность выполнить этот файл. Чтобы выполнить сценарий оболочки непосредственно из командной строки, нужны также права запуска на выполнение. В отличие от этого двоичные (программные) файлы не нужно читать, поскольку они выполняются непосредственным образом. Чтобы запустить двоичный файл на выполнение, нужно иметь только права на его запуск.

Кроме ключей `a` (all — все) и `o` (other — все остальные), в качестве аргументов утилиты chmod можно использовать ключ `g` (group — участники группы) и `u` (user — пользователь, хотя здесь *пользователь* является ссылкой на *владельца* файла, который в каждый конкретный момент времени может быть, а может и не быть пользователем файла). Например, команда `chmod a+x` позволяет добавить права доступа для всех пользователей (для всех остальных, для участников группы и для владельца), а команда `chmod go-rwx` удаляет все права доступа для всех, кроме владельца файла.

CHMOD: КЛЮЧ O — ДЛЯ ВСЕХ ОСТАЛЬНЫХ, А КЛЮЧ U — ДЛЯ ВЛАДЕЛЬЦА

При использовании утилиты chmod многие считают, что ключ «o» означает «owner» — владелец, но это не соответствует действительности. Ключ «o» означает «other» — все остальные, а вот ключ «u» означает «owner» (user) — владелец (пользователь). Акроним UGO (user-group-other, то есть пользователь-группа-все остальные) может помочь вам запомнить обозначения прав доступа в ключах.

Числовые аргументы для утилиты chmod

Чтобы указать права доступа, в утилите chmod можно также использовать числовые аргументы. При этом вместо букв и символов, использовавшихся в предыдущих примерах, используются аргументы, составленные из трех восьмеричных чисел. (Четвертое, лидирующее число, управляет правами доступа setuid и setgid и будет рассмотрено ниже.) Первое число определяет права доступа для владельца, второе — для группы, а третье — для всех остальных. Число 1 предоставляет указанному пользователю (или пользователям) права запуска на выполнение, число 2 — права на запись, а число 4 — права на чтение. Для получения числа, представляющего права доступа для владельца, для группы или для всех остальных, нужно использовать в отношении соответствующих чисел операцию ИЛИ (добавление), как показано в следующих примерах. Использование числовых аргументов позволяет установить права доступа к файлу абсолютным образом, этот способ не модифицирует существующие права доступа, как это было при использовании символьных аргументов.

В следующем примере утилита chmod изменяет права доступа таким образом, что права на чтение файла и на запись в него получает только владелец файла, независимо от тех прав доступа, которые были установлены ранее. Число 6 в первой позиции дает

владельцу права на чтение (4) и запись (2). Числа 0 удаляют все права доступа для группы и всех остальных пользователей.

```
$ chmod 600 letter.0610
$ ls -l letter.0610
-rw------- 1 max pubs 3355 Jun 22 12:44 letter.0610
```

Затем число 7 (4 + 2 + 1) дает владельцу права на чтение, на запись и на запуск на выполнение. А число 5 (4 + 1) дает группе и всем остальным права на чтение и на запуск на выполнение:

```
$ chmod 755 check_spell
$ ls -l check_spell
-rwxr-xr-x 1 max pubs 852 Jul 31 13:47 check_spell
```

Дополнительные примеры использования числовых аргументов для установки прав доступа приведены в табл. V.8. Дополнительные сведения по использованию утилиты chmod для придания файлам сценариев возможности исполнения приведены в разделе «chmod: Превращение файла в исполняемый» на с. 292, а информация, касающаяся не только абсолютных аргументов, но и всей утилиты chmod, приведена в части V.

Права setuid и setgid

При выполнении файла, имеющего установленное setuid-право (set user ID — установить идентификатор пользователя), процесс, выполняющий файл, получает права владельца файла. Например, если вы запускаете setuid-программу, удаляющую все файлы, имеющиеся в каталоге, вы можете удалить файлы в любом из каталогов владельца файла, даже если у вас в обычных условиях нет на это прав. Точно так же файл, имеющий установленное setgid-право (set group ID — установить идентификатор группы), предоставляет процессу, выполняющему файл, права группы, с которой связан этот файл.

ИСПОЛЬЗОВАНИЕ SETUID- SETGID-ПРОГРАММ, ВЛАДЕЛЬЦЕМ КОТОРЫХ ЯВЛЯЕТСЯ ПОЛЬЗОВАТЕЛЬ С ИМЕНЕМ ROOT, НУЖНО СВЕСТИ К МИНИМУМУ

Исполняемые файлы с установленным setuid-правом, владельцем которых является привилегированный пользователь — root, при запуске их на выполнение обладают root-правами, даже если они запущены не root-пользователем. Программа этого типа обладает весьма широкими полномочиями, поскольку она может делать все, на что имеет права root-пользователь (в соответствии с конструктивным предназначением самой программы). Точно так же широкими правами доступа обладают и исполняемые файлы, имеющие setgid-право, владельцем которых является привилегированный пользователь (root).

В силу того, что setuid-программы, имеющие привилегированного владельца (root), и setgid-программы, принадлежащие группе root, обладают широкими правами и мощным разрушительным потенциалом, будет разумнее избегать создания и использования таких программ без крайней необходимости. Из-за опасности, исходящей от этих программ, многие сайты сводят их использование к минимуму. Примером необходимых setuid-программ может послужить утилита passwd.

В следующем примере показывается пользователь, работающий с root-правами и использующий символьные аргументы для утилиты chmod, чтобы установить одной программе setuid-право, а другой программе setgid-право. Информация, выводимая командой `ls -l` (см. раздел «ls –l: Отображение прав» на с. 115), показывает наличие

setuid-права путем отображения в позиции права запуска на выполнение, принадлежащего владельцу, символа s, а наличие setgid-права — путем отображения этого же символа в позиции права запуска на выполнение, принадлежащего группе:

```
$ ls -l myprog*
-rwxr-xr-x 1 root pubs 19704 Jul 31 14:30 myprog1
-rwxr-xr-x 1 root pubs 19704 Jul 31 14:30 myprog2
# chmod u+s myprog1
# chmod g+s myprog2
$ ls -l myprog*
-rwsr-xr-x 1 root pubs 19704 Jul 31 14:30 myprog1
-rwxr-sr-x 1 root pubs 19704 Jul 31 14:30 myprog2
```

Для осуществления таких же изменений в следующем примере с утилитой chmod используются числовые аргументы. Когда для задания прав используются четыре числа, то установка значения первого из них в 1 приводит к появлению единицы в нерабочем бите, установка его значения в 2 задает setgid-права, а установка его значения в 4 задает setuid-права:

```
$ ls -l myprog*
-rwxr-xr-x 1 root pubs 19704 Jul 31 14:30 myprog1
-rwxr-xr-x 1 root pubs 19704 Jul 31 14:30 myprog2
# chmod 4755 myprog1
# chmod 2755 myprog2
$ ls -l myprog*
-rwsr-xr-x 1 root pubs 19704 Jul 31 14:30 myprog1
-rwxr-sr-x 1 root pubs 19704 Jul 31 14:30 myprog2
```

НЕ СОЗДАВАЙТЕ СЦЕНАРИИ ОБОЛОЧКИ С SETUID-ПРАВАМИ

Никогда не устанавливайте для сценариев оболочки setuid-права, поскольку существуют хорошо известные методики использования таких сценариев в целях нанесения вреда системе.

Права доступа к каталогам

Применительно к каталогами права доступа имеют немного иное назначение. Наряду с тем, что три категории пользователей могут считывать данные из каталогов и записывать их в каталоги, они не могут запускать каталоги на выполнение. Возможность запуска на выполнение для каталогов переопределена: она означает возможность назначить каталог в качестве рабочего и (или) просматривать имеющиеся в каталоге файлы, для которых имеются права на чтение. А к выполнению файла каталога это право не имеет никакого отношения.

Когда для каталога имеется право только на выполнение, можно использовать утилиту ls для вывода списка файлов, имеющихся в каталоге. Для вывода списка всего содержимого каталога утилиту ls можно использовать без аргументов. В следующем примере пользователь Zach сначала проверяет тот факт, что он вошел в систему под своим именем, а затем он проверяет наличие прав на каталог info, принадлежащий пользователю Max. Права доступа к каталогу можно просмотреть, запустив утилиту ls с ключами -d (directory — каталог) и -l (long — развернутый формат):

```
$ who am i
zach     pts/7   Aug 21 10:02
```

```
$ ls -ld /home/max/info
drwx-----x   2 max pubs 512 Aug 21 09:31 /home/max/info
$ ls -l /home/max/info
ls: /home/max/info: Permission denied    Отсутствие прав доступа
```

Символ d в крайней левой позиции строки, которую выводит утилита ls, показывает, что /home/max/info — каталог. У пользователя Max имеются права на чтение, запись и выполнение; пользователи, имеющие отношение к группе pubs, не имеют прав доступа, а все остальные пользователи имеют только право на выполнение, о чем свидетельствует символ x в крайней правой позиции строки прав. Поскольку пользователь Zach не имеет для этого каталога права на чтение, команда ls -l возвращает сообщение об ошибке.

Когда Zach указывает имена файлов, информацию о которых он хочет получить, он не считывает информацию о новом каталоге, а просто ищет определенную информацию, что ему разрешено делать при наличии прав запуска каталога на выполнение. У него есть права на чтение файла notes, поэтому с использованием утилиты cat для отображения содержимого файла проблем не возникает. Но он не может вывести содержимое файла financial, поскольку у него нет прав на чтение этого файла:

```
$ ls -l /home/max/info/financial /home/max/info/notes
-rw-------   1 max pubs 34 Aug 21 09:31 /home/max/info/financial
-rw-r--r--   1 max pubs 30 Aug 21 09:32 /home/max/info/notes
$ cat /home/max/info/notes
Это файл по имени notes.
$ cat /home/max/info/financial
cat: /home/max/info/financial: Permission denied    Отсутствие прав доступа
```

После этого пользователь Max дает всем остальным право чтения принадлежащего ему каталога info:

```
$ chmod o+r /home/max/info
```

Когда пользователь Zach проверяет свои права доступа к info, он обнаруживает, что у него есть право на чтение этого каталога и право на его выполнение. Теперь команда ls -l может вывести список содержимого каталога info, но Zach по-прежнему не может читать файл financial. (Это ограничение связано с правом доступа к файлу, а не к каталогу.) И наконец, Zach пытается создать файл по имени newfile, используя утилиту touch (см. часть V). Если бы Max дал ему право на запись в каталог info, Zach смог бы создать в нем новый файл:

```
$ ls -ld /home/max/info
drwx---r-x   2 max pubs 512 Aug 21 09:31 /home/max/info
$ ls -l /home/max/info
total 8
-rw-------   1 max pubs 34 Aug 21 09:31 financial
-rw-r--r--   1 max pubs 30 Aug 21 09:32 notes
$ cat /home/max/info/financial
cat: financial: Permission denied    Отсутствие прав доступа
$ touch /home/max/info/newfile
touch: cannot touch '/home/max/info/newfile': Permission denied
                                     Отсутствие прав доступа
```

Списки управления доступом (ACL-списки)

Списки управления доступом (ACL-списки) предоставляют более тонкую организацию управления доступом пользователей к указанным каталогам и файлам по сравнению с традиционными правами доступа Linux. При использовании ACL-списков можно указать способы, позволяющие каждому из нескольких пользователей иметь доступ к каталогу или файлу. Поскольку ACL-списки могут снизить производительность системы, их использование не следует разрешать на файловых системах, хранящих системные файлы, где вполне достаточно использовать традиционные права доступа Linux. Нужно также быть осмотрительным при перемещении, копировании или архивировании файлов, поскольку не все утилиты сохраняют ACL-списки. Кроме того, скопировать ACL-списки в файловые системы, которые их не поддерживают, невозможно.

ACL-список включает в себя набор правил. Правило определяет, как указанный пользователь или группа может получить доступ к файлу, с которым связан ACL-список. Есть два вида правил: *правила доступа* и *правила по умолчанию*. (В документации имеются ссылки на *ACL-списки доступа* (access ACLs) и на *ACL-списки по умолчанию* (default ACLs), несмотря на то что есть только один вид ACL-списка и два вида правил, которые может содержать в себе ACL-список.)

Правило доступа определяет информацию о доступе для отдельного файла или каталога. ACL-список по умолчанию принадлежит исключительно каталогу, он определяет информацию доступа по умолчанию (ACL-список) для любого, имеющегося в каталоге файла, которому не предоставлен явно заданный ACL-список.

БОЛЬШИНСТВО УТИЛИТ НЕ СОХРАНЯЕТ ACL-СПИСКИ

Когда утилита cp используется с ключами –p (preserve — сохранить) или –a (archive — архивировать), она сохраняет ACL-списки при копировании файлов. Утилита mv также сохраняет ACL-списки. В случае когда утилита cp используется с ключом –p или с ключом –a и не в состоянии скопировать ACL-списки, а также в случае, когда утилита mv не может сохранить ACL-списки, утилита выполняет свою операцию и выдает сообщение об ошибке:

```
$ mv report /tmp
mv: preserving permissions for '/tmp/report': Operation not supported
(mv: сохранение прав доступа для  '/tmp/report': Операция не поддерживается)
```

Другие утилиты, такие как tar, cpio и dump, не поддерживают ACL-списки. А использование утилиты cp с ключом –a для копирования иерархии каталогов, включающих ACL-списки, не представляется возможным.

Также невозможно скопировать ACL-списки в файловую систему, которая их не поддерживает, или в файловую систему, в которой выключена поддержка этих списков.

Предоставление возможности использования ACL-списков

Следующее объяснение способа предоставления возможности использования ACL-списков относится исключительно к системе Linux.

Перед тем как получить возможность использования ACL-списков, нужно установить программный пакет acl.

Формально Linux поддерживает ACL-списки только на файловых системах ext2, ext3 и ext4, хотя неформально поддержка ACL-списков доступна и на других фай-

ловых системах. Для использования ACL-списков на файловой системе ext нужно смонтировать устройство с ключом acl (по умолчанию используется ключ no_acl). Например, если нужно смонтировать устройство, представленное файлом /home, чтобы использовать ACL-списки применительно к файлам, находящимся в /home, можно добавить acl к списку ключей в файле /etc/fstab:

```
$ grep home /etc/fstab
LABEL=/home             /home           ext3    defaults,acl        1 2
```

После внесения изменений в файл fstab и перед тем как воспользоваться возможностью использования ACL-списков, нужно будет перемонтировать /home. Если систему не использует кто-нибудь еще, вы можете демонтировать это устройство и смонтировать его заново (работая с root-правами), если только в качестве рабочего не используется какой-нибудь каталог в иерархии /home. Кроме этого, для размонтирования /home в процессе использования устройства можно с утилитой mount использовать ключ remount:

```
# mount -v -o remount /home
/dev/hda3 on /home type ext3 (rw,acl)
```

Работа с правами доступа

Утилита setfacl модифицирует ACL-список файла, а утилита getfacl отображает этот список. Эти утилиты доступны только при работе в системе Linux. При использовании утилиты getfacl для получения информации о файле, который не имеет ACL-списка, она показывает ту же информацию, что и команда ls -l, хотя и несколько в ином формате:

```
$ ls -l report
-rw-r--r--  1 max max 9537 Jan 12 23:17 report
$ getfacl report
# file: report
# owner: max
# group: max
user::rw-
group::r--
other::r--
```

Первые три строки информации, выводимой утилитой getfacl, содержат заголовок. В них указывается имя файла, его владелец и группа, с которой связан файл. Дополнительные сведения можно найти в разделе «ls –l: Отображение прав» на с. 115. Ключ --omit-header (или просто -omit — опустить) заставляет утилиту getfacl не показывать заголовок:

```
$ getfacl --omit-header report
user::rw-
group::r--
other::r--
```

В строке, которая начинается со слова user, два двоеточия (::) без имени между ними указывают на то, что в строке показываются права владельца файла. Точно так же

два двоеточия в строке group указывают на то, в этой строке показываются права той группы, с которой связан файл. Два двоеточия, идущие за словом other, используются в целях совместимости: со всеми остальными пользователями (other) не могут быть связаны никакие имена.

При использовании утилиты setfacl ключ -- modify (или -m) позволяет добавить или изменить одно или несколько правил в ACL-списке файла, применив следующий формат:

```
setfacl --modify ugo:имя:права список_файлов
```

где *ugo* может принимать значение u, g или o, чтобы указать, что команда устанавливает права для пользователя (владельца), группы или всех остальных пользователей соответственно; *имя* — это имя пользователя или группы, для которой устанавливаются права; *права* — это права в символьном или числовом формате; и *список_файлов* — это файлы, к которым будут применяться эти права. При указании прав для всех остальных пользователей (o), *имя* нужно опустить. При указании прав в символьном формате для отображения прав доступа используются буквы (rwx, r-x и т. д.), а при задании абсолютных прав используется восьмеричное число. При использовании утилиты chmod применяются три набора прав или три восьмеричных числа (по одному для владельца, группы и всех остальных пользователей), а при использовании утилиты setfacl применяется всего один набор прав или одно восьмеричное число для предоставления прав пользователю или группе, на которых указывают элементы *ugo* и *имя*. Дополнительные сведения о символьном или абсолютном представлении прав доступа к файлу можно найти в разделе, посвященном утилите chmod.

Например, обе следующие команды добавляют правило к ACL-списку, которое дает пользователю Sam права на чтение файла и на запись в него данных:

```
$ setfacl --modify u:sam:rw- report
```

или

```
$ setfacl --modify u:sam:6 report
$ getfacl report
# file: report
# owner: max
# group: max
user::rw-
user:sam:rw-
group::r--
mask::rw-
other::r--
```

Строка, содержащая текст user:sam:rw-, показывает, что пользователь Sam имеет доступ к файлу по чтению и записи (rw-). Порядок чтения прав доступа изложен в разделе «ls –l: Отображение прав» на с. 115. Описание строки, которая начинается со слова mask, дано в следующем дополнительном разделе.

Когда у файла есть ACL-список, команда ls -l отображает знак плюс (+), который следует за строкой прав, даже если ACL-список пуст:

```
$ ls -l report
-rw-rw-r--+ 1 max max 9537 Jan 12 23:17 report
```

Маска эффективных прав

Строка, начинающаяся со слова mask, указывает на маску эффективных прав. Эта маска ограничивает эффективные права, выделенные с помощью ACL-списков группам и пользователям. Она не затрагивает владельца файла или группу, с которой связан файл. Иными словами, она не оказывает влияние на традиционные права доступа Linux. Но, поскольку утилита setfacl всегда устанавливает маску эффективных прав на минимальные ограничительные права доступа к файлу, устанавливаемые в ACL-списке, эта маска не оказывает никакого влияния до тех пор, пока она не будет установлена явным образом после установки для файла ACL-списка. Маску можно установить, указав «mask» вместо «ugo» и не указывая «имя» в команде setfacl.

В следующем примере устанавливается маска эффективных прав на чтение для файла report:

```
$ setfacl -m mask::r-- report
```

В строке mask в следующей информации, выводимой утилитой getfacl, показана маска эффективных прав, установленная на чтение (r- -). Строка, отображающая права пользователя Sam на доступ к файлу, показывает, что они по-прежнему распространяются на чтение и запись. Но комментарий в правой половине строки показывает, что его эффективные права распространяются только на чтение.

```
$ getfacl report
# file: report
# owner: max
# group: max
user::rw-
user:sam:rw-                    #effective:r--
group::r--
mask::r--
other::r--
```

Как показывает следующий пример, утилита setfacl может изменить ACL-правила и может одновременно установить более одного ACL-правила:

```
$ setfacl -m u:sam:r--,u:zach:rw- report
$ getfacl --omit-header report
user::rw-
user:sam:r--
user:zach:rw-
group::r--
mask::rw-
other::r--
```

Ключ -x приводит к удалению ACL-правил для пользователя или группы. Он не оказывает влияния на права владельца файла или права группы, с которой связан этот файл. В следующем примере показана утилита setfacl, удаляющая правило, которое дает пользователю Sam право доступа к файлу:

```
$ setfacl -x u:sam report
$ getfacl --omit-header report
user::rw-
user:zach:rw-
group::r--
mask::rw-
other::r--
```

При использовании ключа -x не нужно указывать *права*. Указываются только *ugo* и *имя*. Ключ -b, за которым следует только лишь имя файла, приводит к удалению ACL-правил и самого ACL-списка из указанного файла или каталога.

С утилитами setfacl и getfacl используется много ключей. Чтобы вывести краткий список ключей или ссылку на man-страницы, используется ключ `--help`.

Установка правил по умолчанию для каталога

В следующем примере показывается, что каталог dir изначально не имел ACL-списка. В команде setfacl используется ключ `-d` (default — по умолчанию), чтобы добавить два правила по умолчанию к ACL-списку для каталога dir. Эти правила применяются ко всем файлам, имеющимся в каталоге dir, у которых нет явно установленных ACL-списков. Правила дают участникам группы `pubs` права на чтение и выполнение и дают права участникам группы `admin` права на чтение, запись и выполнение.

```
$ ls -ld dir
drwx------ 2 max max 4096 Feb 12 23:15 dir
$ getfacl dir
# file: dir
# owner: max
# group: max
user::rwx
group::---
other::---
$ setfacl -d -m g:pubs:r-x,g:admin:rwx dir
```

Следующая команда ls показывает, что у каталога dir теперь есть ACL-список, о чем свидетельствует знак + справа от строки прав доступа. Каждое из правил по умолчанию, которое отображает утилита getfacl, начинается со слова `default:`. Первые два правила по умолчанию и последнее правило по умолчанию определяют права доступа для владельца файла, для группы, с которой связан этот файл, и для всех остальных пользователей. Эти три правила определяют традиционные права доступа Linux и имеют приоритет над другими ACL-правилами. Третье и четвертое правила определяют права доступа для групп `pubs` и `admin`. Далее следует маска эффективных прав.

```
$ ls -ld dir
drwx------+ 2 max max 4096 Feb 12 23:15 dir
$ getfacl dir
# file: dir
# owner: max
# group: max
user::rwx
group::---
other::---
default:user::rwx
default:group::---
default:group:pubs:r-x
default:group:admin:rwx
default:mask::rwx
default:other::---
```

Следует помнить, что правила по умолчанию относятся к файлам, которые содержатся в каталоге и которым ACL-списки не присвоены явным образом. Вы можете также определить правила доступа для самого каталога.

При создании файла в каталоге, который в своем ACL-списке содержит правила по умолчанию, маска эффективных прав для этого файла создается на основе прав

доступа к этому файлу. В некоторых случаях маска может перезаписать ACL-права по умолчанию.

В следующем примере утилита touch создает файл по имени new в каталоге dir. Команда ls показывает, что у этого файла есть ACL-список. На основе значения маски umask (см. man-страницу, посвященную оболочке bash) и владелец, и группа, с которой связан файл, имеют права на чтение данных из файла и запись их в него. Маска эффективных прав устанавливается на чтение и запись таким образом, чтобы для группы pubs выделялись эффективные права на чтение, а для группы admin — на чтение и запись. Ни одна из групп не получает права на выполнение.

```
$ cd dir
$ touch new
$ ls -l new
-rw-rw----+ 1 max max 0 Feb 13 00:39 new
$ getfacl --omit new
user::rw-
group::---
group:pubs:r-x                  #effective:r--
group:admin:rwx                 #effective:rw-
mask::rw-
other::---
```

Если изменить традиционные права на чтение, запись и выполнение для владельца и группы, маска эффективных прав изменится на чтение, запись и выполнение, и группы, определенные правами по умолчанию, получат права на выполнение файла.

```
$ chmod 770 new
$ ls -l new
-rwxrwx---+ 1 max max 0 Feb 13 00:39 new
$ getfacl --omit new
user::rwx
group::---
group:pubs:r-x
group:admin:rwx
mask::rwx
other::---
```

Ссылки

Ссылка является указателем на файл. При создании файла с использованием утилит vim, touch, cp или других средств вы помещаете указатель в каталог. Этот указатель связывает имя файла с местом на диске. Когда в команде указывается имя файла, вы опосредованно указываете на то место на диске, в котором содержится нужная вам информация.

Когда двое и более людей работают над одним проектом и нуждаются в совместном использовании некоторой информации, они могут воспользоваться файлами коллек-

тивного доступа. Можно облегчить другим пользователям допуск к одному из ваших файлов, создав для этого файла дополнительные ссылки.

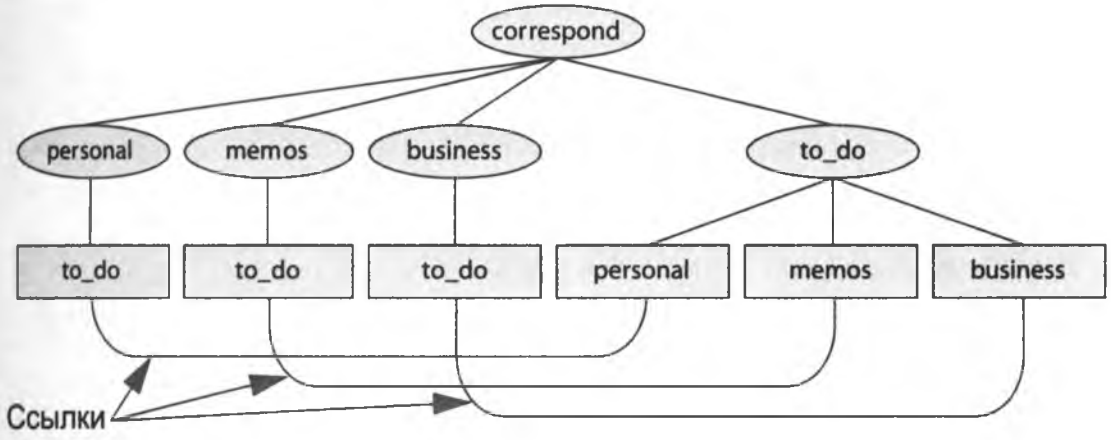

Рис. 4.10. Использование ссылок для перекрестной классификации файлов

Чтобы использовать файл совместно с другим пользователем, сначала нужно дать ему права на чтение из этого файла и на запись в него. Может также понадобиться изменить права доступа к родительскому каталогу этого файла, чтобы дать пользователю права на чтение, запись и выполнение. При соответствующем предоставлении прав пользователь может создать ссылку на файл, чтобы каждый из вас мог получить доступ к файлу со своей отдельной иерархии каталогов.

Ссылка может также оказаться полезной и для отдельного пользователя, имеющего обширную иерархию каталогов. Можно создавать ссылки для перекрестной классификации файлов в вашей иерархии каталогов, при которой для разных задач используются разные классификации. Например, если есть файловая структура, показанная на рис. 4.2, то файл с рабочим планом по имени to_do может появиться в любом подкаталоге соответствующего каталога, то есть personal, memos и business. Если трудно следить за всеми неотложными делами, можно создать в соответствующем каталоге отдельный каталог по имени to_do. Затем в этом каталоге можно создать ссылки на каждый список неотложных дел, имеющийся в подкаталогах. Например, можно связать файл по имени to_do в каталоге memos с файлом по имени memos в каталоге to_do. Этот набор ссылок показан на рис. 4.10.

Несмотря на кажущуюся сложность, эта технология позволяет воспользоваться удобствами содержания всех списков неотложных дел в одном месте. Доступ к соответствующему списку легко можно получить в каталоге, связанном с той или иной задачей, занимаясь составлением писем, написанием служебных записок или своими внутренними делами.

О ДИСКУССИИ ВОКРУГ ЖЕСТКИХ ССЫЛОК

Существует два вида ссылок: жесткие и символьные (мягкие) ссылки. Жесткие ссылки имеют более давнюю историю и считаются устаревшими. Раздел о жестких ссылках помечен дополнительным, и вы можете его пропустить, хотя в нем рассматриваются i-узлы и приводятся некоторые понятия, касающиеся файловой системы.

Жесткие ссылки

Жесткая ссылка на файл появляется в виде другого файла. Если это файл появляется в том же самом каталоге, что и файл, на который делается ссылка, имена файлов-ссылок должны быть другими, поскольку два файла в одном и том же каталоге не могут иметь одинаковые имена. Жесткую ссылку на файл можно создать только из той файловой системы, в которой хранится файл.

ln: Создание жесткой ссылки

Утилита ln (link — ссылка) (без ключа –s или --symbolic) создает жесткую ссылку на существующий файл, при этом используется следующий синтаксис:

```
ln существующий_файл новая_ссылка
```

В следующей команде показано, как пользователь Zach создает новую ссылку (рис. 4.11) по имени /home/max/letter на существующий файл по имени draft в личном каталоге пользователя Zach:

```
$ pwd
/home/zach
$ ln draft /home/max/letter
```

Новая ссылка появляется в каталоге /home/max в файле с именем letter. На практике пользователю Max может понадобиться изменить права доступа к каталогу, чтобы пользователь Zach смог создать ссылку. Даже при том, что файл /home/max/letter появляется в каталоге, принадлежащем пользователю Max, Zach является владельцем файла, потому что он его создал.

Утилита ln создает дополнительный указатель на существующий файл, но она не создает еще одну копию файла. Поскольку файл всего один, информация о статусе файла, например о правах доступа к нему, владельце и времени его последней модификации, для всех ссылок одинакова, отличается только имя файла. Когда, к примеру, пользователь Zach изменяет файл /home/zach/draft, Max видит эти изменения в файле /home/max/letter.

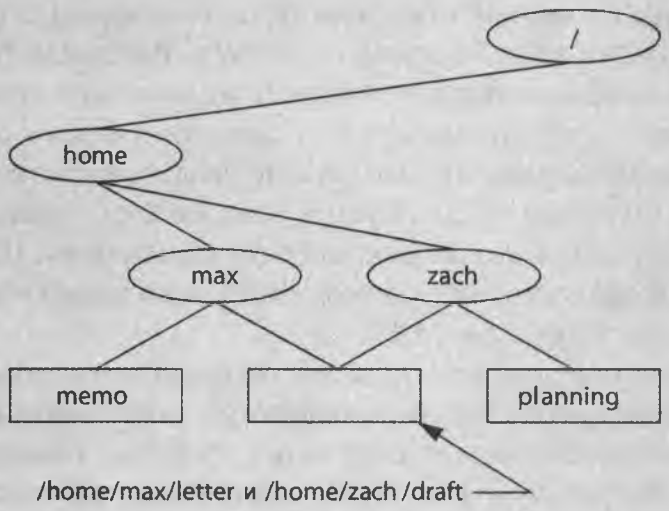

Рис. 4.11. Две ссылки на один и тот же файл: /home/max/letter и /home/zach/draft

Сравнение утилит cp и ln

Следующие команды подтверждают, что утилита ln не создала еще одной копии файла. Создайте файл, воспользуйтесь утилитой ln для создания дополнительной ссылки на него, измените содержимое файла с использованием одной ссылки и проверьте наличие изменений с помощью другой ссылки:

```
$ cat file_a
Это файл A.
```

```
$ ln file_a file_b
$ cat file_b
Это файл A.
$ vim file_b
...
$ cat file_b
Это файл B после изменения.
$ cat file_a
Это файл B после изменения.
```

Если провести такой же эксперимент с использованием вместо ln утилиты cp и изменить копию файла, разница в работе этих двух утилит станет понятнее. Как только копия файла будет изменена, эти два файла станут разными по содержанию:

```
$ cat file_c
Это файл C.
$ cp file_c file_d
$ cat file_d
Это файл C.
$ vim file_d
...
$ cat file_d
Это файл D после изменения.
$ cat file_c
Это файл C.
```

Утилита ls и количество ссылок

Утилиту ls можно использовать с ключом -l, за которым следует имя файла, который нужно сравнить, чтобы подтвердить, что информация о статусе для двух ссылок на один и тот же файл одна и та же и отличается для тех файлов, которые не связаны друг с другом ссылкой. В следующем примере цифра 2 отображается в поле ссылок (слева от max) и показывает, что есть две ссылки на file_a и file_b (из предыдущего примера):

```
$ ls -l file_a file_b file_c file_d
-rw-r--r-- 2 max pubs 33  May 24 10:52 file_a
-rw-r--r-- 2 max pubs 33  May 24 10:52 file_b
-rw-r--r-- 1 max pubs 16  May 24 10:55 file_c
-rw-r--r-- 1 max pubs 33  May 24 10:57 file_d
```

Хотя нетрудно догадаться, какой из файлов в этом примере является ссылкой на другой файл, утилита ls об этом не сообщает.

Утилита ls и i-узлы

Чтобы определить без всяких сомнений, на какой из файлов имеется ссылка, нужно воспользоваться утилитой ls с ключом -i. Этот ключ выводит список количества *i-узлов* для каждого файла. I-узел является управляющей структурой для файла. (HFS+, файловая система, используемая в Mac OS X по умолчанию, не имеет i-узлов, но если применить к ней более сложную схему, то окажется, что i-узлы все же имеются.) Если для двух имен файлов указывается один и тот же номер i-узла, это значит, что они совместно используют одну и ту же управляющую структуру и ссылаются на один и тот же файл. И наоборот, когда для двух имен файлов указываются разные номера i-узлов, то это означает, что мы имеем дело с разными файлами. В следующем примере показывается, что у file_a и file_b один и тот же номер i-узла, а у файлов file_c и file_d разные номера i-узлов:

```
$ ls -i file_a file_b file_c file_d
3534 file_a    3534 file_b    5800 file_c    7328 file_d
```

Все ссылки на файл абсолютно равнозначны: операционная система не способна разобраться в том порядке, в котором создавались несколько ссылок. Когда на файл имеется две ссылки, то любую из них можно удалить и по-прежнему иметь доступ к файлу по оставшейся ссылке. Можно, к примеру, удалить ссылку, которая использовалась для создания файла, и, поскольку осталась еще одна ссылка, иметь доступ к файлу через нее.

Символьные ссылки

Кроме жестких ссылок Linux поддерживает *символьные ссылки*, которые также называют *мягкими* (soft links или symlinks). Жесткая ссылка является указателем на файл (записью в каталоге, указывающей на i-узел), а символьная ссылка — это косвенный указатель на файл (запись в каталоге, которая содержит путевое имя того файла, на который она указывает, то есть она является указателем на жесткую ссылку на файл).

Преимущества символьных ссылок

Символьные ссылки были разработаны из-за ограничений, присущих жестким ссылкам. Нельзя создать жесткую ссылку на каталог, но можно создать на него символьную ссылку.

В большинстве случаев файловая иерархия Linux охватывает сразу несколько файловых систем. Поскольку каждая файловая система содержит отдельную управляющую информацию (то есть отдельные таблицы i-узлов или структуры файловой системы) для тех файлов, которые в ней содержатся, создание жестких ссылок между файлами, принадлежащими разным файловым системам, не представляется возможным. Символьная ссылка может указывать на любой файл, независимо от того, где именно он находится в файловой структуре, а жесткая ссылка на файл должна быть в той же файловой системе, что и другая жесткая ссылка (или ссылки) на файл. При создании ссылок на файлы, которые находятся в вашем каталоге, вы этих ограничений не заметите.

Главное преимущество символьной ссылки — она может указывать на несуществующий файл. Этой возможностью можно воспользоваться, если нужна ссылка на файл, который периодически удаляется или заново создается. Жесткая ссылка продолжает указывать на «удаленный» файл, ссылка на который поддерживается даже после того, как создан новый файл. В отличие от этого символьная ссылка всегда указывает на только что созданный файл и не мешает удалить старый файл. Символьная ссылка может указывать на файл, подвергаемый контролю на входе и выходе системой управления исходным кодом, то есть на файл с расширением .o, который создается компилятором языка C при каждом запуске утилиты make, или на регистрационный файл, который многократно подвергается архивации.

При всей своей универсальности по сравнению с жесткими ссылками символьные ссылки также не избавлены от ряда недостатков. У всех жестких ссылок на файл один и тот же статус, а у символьных ссылок нет такого же статуса, как у жестких ссылок. Когда у файла имеется несколько жестких ссылок, его можно сравнить с человеком, имеющим несколько вполне законных имен, как это бывает у многих замужних женщин. В отличие от этого символьные ссылки сродни прозвищам. У каждого может быть одно или несколько прозвищ, но эти прозвища имеют куда более низкий статус по сравнению с законными именами. В следующих разделах рассматриваются некоторые особенности символьных ссылок.

ln: Создание символьных ссылок

Символьная ссылка создается утилитой ln с ключом `--symbolic` (или `-s`). В следующих примерах создается символьная ссылка /tmp/s3 на файл sum, который находится в личном каталоге пользователя Max. Когда для просмотра символьной ссылки используется команда `ls -l`, утилита ls выводит имя ссылки и имя файла, на который она указывает. В таком случае первым в строке списка стоит символ `l` (link — ссылка).

```
$ ln --symbolic /home/max/sum /tmp/s3
$ ls -l /home/max/sum /tmp/s3
```

```
-rw-rw-r--   1 max max 38 Jun 12 09:51 /home/max/sum
lrwxrwxrwx   1 max max 14 Jun 12 09:52 /tmp/s3 -> /home/max/sum
$ cat /tmp/s3
Это содержимое файла sum.
```

Размеры и показания времени последней модификации у этих двух файлов различаются. В отличие от жесткой ссылки символьная ссылка на файл не имеет той же информации о статусе, что и сам файл.

ИСПОЛЬЗУЙТЕ В СИМВОЛЬНЫХ ССЫЛКАХ АБСОЛЮТНЫЕ ПУТЕВЫЕ ИМЕНА

Символьные ссылки имеют буквальный смысл, и при их создании не используются сведения о каталогах. Ссылка, указывающая на относительное путевое имя, включая простое имя файла, предполагает, что относительное путевое имя имеет отношение к тому каталогу, в котором была создана ссылка (а не к тому каталогу, из которого она была создана). В следующем примере ссылка указывает на файл по имени sum в каталоге /tmp. Поскольку такого файла не существует, утилита cat выводит сообщение об ошибке:

```
$ pwd
/home/max
$ ln --symbolic sum /tmp/s4
$ ls -l sum /tmp/s4
lrwxrwxrwx   1 max     max            3 Jun 12 10:13 /tmp/s4 -> sum
-rw-rw-r--   1 max     max           38 Jun 12 09:51 sum
$ cat /tmp/s4
cat: /tmp/s4: No such file or directory
                (Такой файл или каталог не существует)
```

Команда cd и символьные ссылки

Когда символьная ссылка используется в качестве аргумента команды cd для смены рабочего каталога, результаты могут быть обескураживающими, особенно если вы не понимаете, что используете символьную ссылку.

Если команда cd используется для смены рабочего каталога на тот, что представлен символьной ссылкой, встроенная в оболочку команда pwd (см. главу 5, раздел «Встроенные команды») выводит имя символьной ссылки. А утилита pwd (/bin/pwd) выводит имя каталога, на который указывает ссылка, а не имя ссылки, независимо от того, как вы попали в этот каталог.

```
$ ln -s /home/max/grades /tmp/grades.old
$ pwd
/home/max
$ cd /tmp/grades.old
$ pwd
/tmp/grades.old
$ /bin/pwd
/home/max/grades
```

Когда рабочим снова делается родительский каталог, вы все равно оказываетесь в каталоге, в котором находится символьная ссылка:

```
$ cd ..
$ pwd
/tmp
$ /bin/pwd
/tmp
```

При работе с Mac OS X /tmp является символьной ссылкой на /private/tmp. После запуска OS X, в результате ввода команды cd .. из предыдущего примера, рабочим становится каталог /private/tmp.

Утилиту ln можно также использовать для создания символьной ссылки на каталог. Если воспользоваться ключом `--symbolic`, утилита ln будет работать, как и ожидалось, независимо от того, на какой именно файл создается ссылка, на обычный файл или на файл каталога.

rm: Удаление ссылки

При создании файла на него создается одна жесткая ссылка. Затем вы можете удалить этот файл или, используя терминологию Linux, удалить ссылку, используя утилиту rm. После удаления последней жесткой ссылки на файл операционная система высвобождает пространство, которое занимал этот файл на диске, и доступ к информации, хранившейся в файле, закрывается. Это пространство высвобождается даже тогда, когда остается символьная ссылка на файл. Когда имеется более одной жесткой ссылки на файл, можно удалить жесткую ссылку и по-прежнему иметь доступ к файлу благодаря любой другой оставшейся ссылке. В отличие от DOS и Windows, система Linux не предоставляет простого способа восстановления удаленного файла. Но опытный специалист может иногда собрать файл по кускам, потратив на это достаточно времени и сил.

После удаления всех жестких ссылок на файл получить доступ к нему по символьной ссылке уже не удастся. В следующем примере утилита cat сообщает, что файл вообще не существует, поскольку она имеет дело с символьной ссылкой на удаленный файл:

```
$ ls -l sum
-rw-r--r-- 1 max pubs 981  May 24 11:05 sum
$ ln -s sum total
$ rm sum
$ cat total
cat: total: No such file or directory  Такой файл или каталог не существует
$ ls -l total
lrwxrwxrwx 1 max pubs 6  May 24 11:09 total -> sum
```

При удалении файла нужно убедиться в том, что удалены все символьные ссылки на этот файл. Удаление символьной ссылки осуществляется так же, как и удаление всех остальных файлов:

```
$ rm total
```

Заключение

В Linux используется иерархическая или древовидная структура, позволяющая так организовать файлы, чтобы их можно было легко и просто найти. Файловая структура содержит файлы каталогов и обычные файлы. Каталоги содержат другие файлы, включая и другие каталоги, а обычные файлы в основном содержат текст, программы или изображения. Все файлы берут свое начало из корневого каталога, который представлен отдельно стоящим или расположенным в крайней левой позиции путевого имени символом слэша (/).

Большинство файловых систем Linux поддерживает длину имен файлов до 255 символов. Но несмотря на это, лучше пользоваться простыми и интуитивно понятными именами. Расширения имен файлов способны придать им больше смысла.

При входе в систему вы всегда оказываетесь связанным с рабочим каталогом. С момента входа в систему и до использования команды cd с целью смены рабочего каталога в его качестве выступает ваш личный каталог.

Абсолютное путевое имя начинается с корневого каталога и содержит все имена файлов, по которым прослеживается путь к заданному файлу. Путевое имя начинается с символа слэша, представляющего собой корневой каталог, и содержит дополнительные слэши, следующими за всеми, встречающимися на пути именами каталогов, за исключением последнего каталога, в том случае, если путь указывает на файл каталога.

Относительные путевые имена похожи на абсолютные, но путь в них отслеживается, начиная с рабочего каталога. Простое имя файла является последним элементом путевого имени и является формой относительного путевого имени, представляющего файл в рабочем каталоге.

Файловая система Linux содержит много важных каталогов, включая /usr/bin, в котором находится большинство программ, обеспечивающих выполнение сервисных команд Linux, и /dev, содержащий файлы устройств, многие из которых являются представителями физических составляющих компьютерного оборудования. К важным стандартным файлам относится /etc/passwd, который содержит информацию о пользователях, в частности идентификационные сведения о каждом пользователе и его полное имя.

К атрибутам, связанным с каждым файлом, относятся и права доступа. Они определяют, кто может получить доступ к файлу и в каком объеме. Потенциально доступ к файлу могут получить три группы пользователей: его владелец, участники группы и все остальные пользователи. Доступ к обычному файлу может осуществляться тремя способами: по чтению, по записи и по выполнению. Эти права отображаются с помощью утилиты ls с ключом -l. Для каталогов доступ по выполнению переопределяется и по смыслу означает возможность ведения в них поиска.

Владелец файла или пользователь, работающий с root-правами, может воспользоваться утилитой chmod и изменить права доступа к файлу. Эта утилита определяет права доступа по чтению, записи и выполнению для владельца файла, группы и всех остальных пользователей, работающих в системе.

Списки управления доступом (ACL-списки) предоставляют более тонкую организацию управления доступом пользователей к указанным каталогам и файлам по сравнению с традиционными правами доступа Linux. Использование ACL-списков позволяет указывать права получения доступа к каталогу или файлу для каждого из нескольких пользователей. При работе с файлами сохранность ACL-списков обеспечивается только лишь несколькими утилитами.

В обычном файле хранятся данные, например текстовая информация, программы или изображения. Каталоги представляют собой файлы стандартной формы, хранящиеся на диске, в которых содержится информация, включая имена, касающаяся обычных файлов и других файлов каталогов. I-узлы являются структурой данных, хранящейся на диске, которая определяет существование файлов и идентифицирует их по номерам этих узлов. Каталог связывает каждое из хранящихся в нем имен файлов с i-узлом.

Ссылка является указателем на файл. На один и тот же файл может быть несколько ссылок, что позволяет использовать этот файл совместно с другими пользователями или допускает его появление более чем в одном каталоге. Поскольку при наличии

нескольких ссылок имеется лишь один экземпляр файла, изменения, вносимые в этот файл при использовании одной из этих ссылок, видны при доступе к нему через все остальные ссылки. Жесткие ссылки не могут быть связаны с каталогами или распространяться на несколько файловых систем, но такими возможностями обладают символьные ссылки.

В табл. 4.2 приводится сводка утилит, представленных в данной главе.

Таблица 4.2. Утилиты, представленные в главе 4

Утилита	Функция
cd	Связывает вас с другим рабочим каталогом
chmod	Изменяет права доступа к файлу
getfacl	Отображает принадлежащий файлу ACL-список
ln	Создает ссылку на существующий файл
mkdir	Создает каталог
pwd	Отображает путевое имя рабочего каталога
rmdir	Удаляет каталог
setfacl	Модифицирует принадлежащий файлу ACL-список

Упражнения

1. Относится ли каждый из следующих примеров к абсолютному путевому имени, к относительному путевому имени или к простому имени файла?
   1) milk_co
   2) correspond/business/milk_co
   3) /home/max
   4) /home/max/literature/promo
   5) ..
   6) letter.0610
2. Перечислите команды, которые можно использовать для осуществления следующих операций:
   1) Сделать ваш личный каталог рабочим.
   2) Идентифицировать рабочий каталог.
3. При условии, что рабочим является каталог /home/max, имеющий подкаталог по имени literature, приведите примеры трех наборов команд, которые можно использовать для создания подкаталога по имени classics в каталоге literature. Также приведите примеры наборов команд, которые можно использовать для удаления каталога classics и его содержимого.
4. Утилита df показывает все смонтированные файловые системы, а также информацию о каждой системе. Воспользуйтесь утилитой df с ключом -h (human-readable — в удобной для чтения форме), чтобы ответить на следующие вопросы:
   1) Сколько файловых систем смонтировано на вашей системе Linux?
   2) В какой файловой системе хранится ваш личный каталог?
   3) Если предположить, что на вопрос 4.1 вы дали ответ о наличии двух или более файловых систем, попробуйте создать жесткую ссылку на файл, принадлежащий другой файловой

системе. Какое сообщение об ошибке вы получите? Что произойдет при попытке создать вместо этого символьную ссылку на этот файл?

5. Предположим, у вас есть файл, который ссылается на файл, принадлежащий другому пользователю. Как убедиться в том, что совместное внесение изменений в файл уже недоступно?
6. У вас должны быть права на чтение файла /etc/passwd. Для ответа на следующий вопрос воспользуйтесь утилитой cat или утилитой less, чтобы вывести содержимое файла /etc/passwd. Посмотрите на информационные поля, имеющиеся в файле /etc/passwd, которые относятся к пользователям вашей системы.
   1) Какой символ используется в качестве разделителя полей в файле /etc/passwd?
   2) Сколько полей используется для описания каждого пользователя?
   3) Сколько пользователей зарегистрировано на локальной системе?
   4) Сколько разных оболочек входа в вашу систему в ней используется? (Подсказка: обратите внимание на последнее поле.)
   5) Во втором поле файла /etc/passwd в закодированной форме хранится пароль пользователя. Если в поле пароля хранится символ x, значит, в вашей системе используются теневые пароли, и закодированные пароли хранятся где-то в другом месте. Использует ли ваша система теневые пароли?
7. Если файлы /home/zach/draft и /home/max/letter являются ссылками на один и тот же файл и происходит следующая череда событий, какой будет дата написания письма, фигурирующего в файле letter?
   1) Пользователь Max вводит команду `vim letter`.
   2) Пользователь Zach вводит команду `vim draft`.
   3) Пользователь Zach изменяет дату написания письма на 31 января 2009 года, записывает файл и выходит из редактора vim.
   4) Пользователь Max изменяет дату написания письма на 1 февраля 2009 года, записывает файл и выходит из редактора vim.
8. Предположим, что пользователь является участником группы, обладающей всеми правами на доступ к файлу по имени jobs_list, но, являясь владельцем этого файла, он тем не менее не имеет вообще никаких прав. Опишите, какие операции, если таковые имеются, может производить с файлом jobs_list пользователь-владелец. Какие команды может вводить пользователь-владелец, чтобы предоставить себе самому все права доступа к файлу?
9. Имеет ли корневой каталог какие-нибудь подкаталоги, в которых вы, как обычный пользователь, не можете вести поиск? Есть ли у корневого каталога какие-нибудь подкаталоги, которые вы не можете прочитать будучи обычным пользователем? Дайте развернутый ответ.
10. Предположим, у вас есть структура каталогов, показанная на рис. 4.2, и следующие права доступа к каталогам:

```
d--x--x---   3 zach pubs 512 Mar 10 15:16 business
drwxr-xr-x   2 zach pubs 512 Mar 10 15:16 business/milk_co
```

Что произойдет, если запустить каждую из следующих команд для каждого из представителей следующих категорий владельцев прав — владельца файла, участника группы и всех остальных пользователей? Предположим, что рабочим является родительский каталог подкаталога correspond и что у всех имеются права на чтение файла cheese_co.

   1) cd correspond/business/milk_co
   2) ls –l correspond/business
   3) cat correspond/business/cheese_co

Упражнения повышенной сложности

1. Что такое i-узел? Что происходит с i-узлом при перемещении файла в пределах файловой системы?
2. На что указывает элемент каталога . .? На что указывает этот элемент в корневом (/) каталоге?
3. Как создать файл по имени –i? Какие технологии при этом не работают и почему? Как можно удалить файл по имени –i?
4. Предположим, рабочий каталог содержит всего один файл по имени andor. Какое сообщение об ошибке вы получите, запустив следующую командную строку?

```
$ mv andor and\/or
```

   При каких обстоятельствах можно запустить эту команду, не вызывая ошибки?
5. Команда `ls -i` отображает имя файла, которому предшествует номер i-узла (см. раздел «Утилита ls и i-узлы» на с. 129). Напишите команду на вывод пар i-узел – имя файла, отсортированных по номерам i-узлов, для файлов, находящихся в рабочем каталоге. (Подсказка: воспользуйтесь каналом.)
6. Как вы думаете, имеет ли системный администратор права доступа к программе, декодирующей пароли пользователей? Почему да или почему нет?
7. Можно ли отличить сам файл от жесткой ссылки на файл? То есть располагая именем файла, можете ли вы сказать, был ли он создан с помощью команды `ln`? Объясните свой ответ.
8. Объясните сообщения об ошибках, отображаемые при выполнении следующей последовательности команд:

```
$ ls -l
total 1
drwxrwxr-x   2 max pubs 1024 Mar  2 17:57 dirtmp
$ ls dirtmp
$ rmdir dirtmp
rmdir: dirtmp: Directory not empty   Каталог не пуст
$ rm dirtmp/*
rm: No match.  Соответствий не найдено
```

Глава 5. Оболочка

В этой главе подробно рассматривается оболочка и объясняется использование ряда присущих ей свойств. Здесь дается разбор синтаксиса командной строки и описание того, как оболочка проводит обработку командной строки и инициирует выполнение программы. В дополнение к этому объясняется, как можно перенаправить ввод в команду и вывод из команды, установить в командной строке каналы и фильтры и запустить команду в фоновом режиме. В последнем разделе рассматриваются расширения имен файлов и объясняется, как их использовать на практике.

За исключением специально оговариваемых случаев все, что рассматривается в этой главе, применимо к оболочкам Bourne Again (bash) и TC (tcsh). Разные оболочки могут выводить на экран разные конкретные слова: информация, отображаемая той оболочкой, которой вы пользуетесь, может слегка отличаться от той, которая будет встречаться в данной книге. Сведения, касающиеся конкретных оболочек, изложены в главах 8 (bash) и 9 (tcsh). В главе 10 будет рассмотрено создание и выполнение сценариев оболочки bash.

Командная строка

Когда в ответ на приглашение вводится команда, оболочка выполняет соответствующую этой команде программу. Например, когда вводится команда `ls`, оболочка выполняет служебную программу (утилиту) по имени ls. Можно заставить оболочку выполнять и другие типы программ, такие как сценарии оболочки, прикладные программы и программы, написанные самим пользователем. Строка, в которой содержится команда, включая любые аргументы, называется *командной строкой*. В этой книге термин *команда* употребляется для ссылки как на символы, введенные в командную строку, так и на программу, вызываемую этими действиями.

Синтаксис

Синтаксис командной строки определяет порядок следования и разделения элементов командной строки. При нажатии клавиши ВВОД (после окончания набора команды) оболочка сканирует командную строку, проверяя в ней правильность использованного синтаксиса. Как правило, для командной строки используется следующий синтаксис:

команда [арг1][арг2] ... [аргп] ВВОД

Элементы командной строки должны быть отделены друг от друга одним или несколькими пробелами. *Команда* — это имя команды, *арг1–аргп* — аргументы, а *ВВОД* — это клавиша, нажатие которой завершает ввод командной строки. Квадратные скобки в синтаксисе показывают, что заключенные в них элементы являются необязательными. (Не все команды требуют обязательного использования аргументов: некоторые команды вообще не разрешают их использования, некоторые разрешают использование разного количества аргументов, а некоторые требуют использования вполне

определенного количества аргументов.) Ключи – особая разновидность аргументов, предваряемая одним или двумя знаками дефиса (прочерками, знаками минуса).

Имя команды

Сообщение о порядке использования

Некоторые весьма полезные команды Linux состоят только из имени команды и не содержат аргументов. Например, сама по себе команда ls выводит список содержимого рабочего каталога. При использовании с неверными аргументами или с неверным количеством аргументов команды, которым нужны аргументы, обычно выдают короткие сообщения об ошибках, так называемые *сообщения о порядке использования*.

Аргументы

В командной строке каждая последовательность непробельных символов называется лексемой или словом. *Аргумент* — это лексема, например имя файла, строка текста, число или другой объект, с которым работает команда. Например, аргументом для команды `vim` или `emacs` служит имя того файла, который нужно отредактировать.

В следующей команде показан вызов утилиты cp, копирующей файл по имени temp в файл по имени tempcopy:

```
$ cp temp tempcopy
```

Аргументы нумеруются, начиная с самой команды, которая считается нулевым аргументом. В данном примере `cp` — это нулевой аргумент, `temp` — первый аргумент, а `tempcopy` — второй. Утилита cp требует наличия в командной строке как минимум двух аргументов (см. часть V). Первым аргументом должно быть имя существующего файла, а вторым — имя файла, который будет создан или переписан утилитой cp. Здесь аргументы не являются необязательными: чтобы команда работала, должны присутствовать оба аргумента. Если количество или вид аргументов неправильны, утилита cp выводит сообщение о порядке использования. Чтобы увидеть его, наберите `cp`, а затем нажмите клавишу ВВОД.

Ключи

Ключ — это аргумент, меняющий действие команды. Часто можно задавать сразу несколько ключей, изменяя поведение команды в нескольких направлениях. Ключи имеют специфическое отношение к той программе, которая вызывается из командной строки, и интерпретируются (обрабатываются) не оболочкой, а самой этой программой.

По соглашению, ключи — это отдельные аргументы, которые следуют за именем команды и обычно предшествуют другим аргументам, таким как имена файлов. Большинство утилит требует, чтобы впереди ключа ставился одиночный дефис. Но данное требование специфично для утилиты, а не для оболочки. Ключи GNU-программ часто предваряются строкой из двух дефисов. Например, ключ `--help` приводит к выводу сообщения (порой слишком детального) о порядке использования той или иной утилиты.

В листинге 5.1 сначала показана информация, выводимая командой ls без аргументов. По умолчанию команда ls выводит перечень содержимого рабочего каталога в алфавитном порядке, отображая его вертикально по столбцам. Затем следует ключ

-r (reverse order — в обратном порядке; поскольку это GNU-утилита, то для этого можно также использовать ключ --reverse), заставляющий утилиту ls вывести список файлов в обратном алфавитном порядке, по-прежнему отсортированный по столбцам. Ключ -x заставляет утилиту ls вывести список файлов в горизонтально отсортированных строках.

Листинг 5.1. Использование ключей

```
$ ls
hold   mark  names   oldstuff  temp  zach
house  max   office  personal  test
$ ls -r
zach  temp      oldstuff  names  mark   hold
test  personal  office    max    house
$ ls -x
hold      house     mark  max   names  office
oldstuff  personal  temp  test  zach
$ ls -rx
zach   test  temp  personal  oldstuff  office
names  max   mark  house     hold
```

Составные ключи

При использовании нескольких однобуквенных ключей часто допускается группировка их в один аргумент, который начинается с одиночного дефиса; ставить пробелы между такими ключами не нужно. Составлять ключи, перед которыми ставятся двойные дефисы, этим способом невозможно. Конкретные правила для составления ключей зависят от используемой программы.

В листинге 5.1 показаны оба ключа, -r и -x, используемые с утилитой ls. Взятые вместе, эти ключи приводят к созданию списка имен файлов в горизонтально отсортированных столбцах, в обратном алфавитном порядке. Команда ls -xr выдает такой же результат, что и команда ls -rx. Команда ls -x -r также создает точно такой же список.

Аргументы ключей

У некоторых утилит есть ключи, которые сами по себе требуют использования аргументов. Например, у утилиты gcc есть ключ -o, за которым должно следовать имя, которое вы хотите присвоить исполняемому файлу, создаваемому этой утилитой. Обычно аргумент ключа отделяется от буквы этого ключа пробелом:

```
$ gcc -o prog prog.c
```

ОТОБРАЖЕНИЕ РАЗМЕРОВ ФАЙЛОВ В УДОБНОМ ФОРМАТЕ: КЛЮЧ –H

Большинство утилит, сообщающих о размерах файлов, указывают размер файла в байтах. Это удобно, когда работа ведется с небольшими файлами, но числа могут читаться с трудом, если мы работаем с файлами, объем которых измеряется в мегабайтах или гигабайтах. Пользуйтесь ключом –h (или ключом --human-readable — легко читаемый формат) для отображения размеров файлов в кило-, мега- и гигабайтах. Проэкспериментируйте с командой df –h (disk free — свободное место на диске) и с командой ls –lh.

Аргументы, начинающиеся со знака дефиса

Другое соглашение позволяет утилитам работать с такими аргументами, как имена файлов, начинающиеся со знака дефиса. Если в качестве имени файла используется -l, следующая команда приобретет неоднозначный смысл:

```
$ ls -l
```

Она может означать, что вам нужно вывести развернутый список всех файлов, находящихся в рабочем каталоге, или же вывести сведения о файле по имени –l. Команда интерпретируется по первому варианту. Поэтому старайтесь избегать создания файлов с именами, начинающимися со знака дефиса. Если же такие файлы были созданы, то многие утилиты соответствуют соглашению, по которому аргумент -- (два последовательных дефиса) показывает окончание области ключей (и начало области аргументов). Чтобы лишить предыдущую команду неоднозначности, следует набирать такую командную строку:

```
$ ls -- -l
```

Можно также воспользоваться альтернативным форматом, в котором точка означает ссылку на рабочий каталог, а слэш показывает, что имя относится к файлу, который находится в рабочем каталоге:

```
$ ls ./-l
```

Если предположить, что вы работаете в каталоге /home/max, предыдущая команда является функциональным эквивалентом следующей команды:

```
$ ls /home/max/-l
```

А эта команда отображает развернутую информацию об этом файле:

```
$ ls -l -- -l
```

Все эти соглашения не являются непреложными правилами, и ряд утилит их не придерживается (например, утилита find). Следование подобным соглашениям для собственных программ — разумная идея, упрощающая пользователям работу с вашей программой. При создании сценариев оболочки, требующих применения ключей, нужно придерживаться соглашений по применению ключей в системе Linux. Можно воспользоваться в сценарии оболочки утилитой xargs (см. часть V), которая поможет следовать этим соглашениям.

Обработка командной строки

По мере ввода командной строки имеющийся в Linux драйвер устройства tty (входящий в ядро Linux) проверяет каждый символ, чтобы узнать, не нужно ли ему предпринять какие-либо немедленные действия. После нажатия комбинации клавиш CONTROL+H (для стирания символа) или CONTROL+U (для удаления всей строки) драйвер устройства тут же приводит командную строку в нужное состояние. Оболочка никогда не видит стертый символ (или символы) или удаленную вами строку. Зачастую такие же правки происходят при нажатии комбинации клавиш CONTROL+W (для удаления слова). Когда введенный вами символ не требует мгновенной реакции, драйвер устройства сохраняет его в буфере и ждет дополнительных символов. Когда вы нажимаете клавишу ВВОД, драйвер устройства передает командную строку оболочке для ее последующей обработки.

Разбор командной строки

Когда оболочка обрабатывает командную строку, она рассматривает ее целиком, а затем производит ее разбор (разбиение) на составные части (рис. 5.1). Затем оболочка ищет имя команды. Обычно имя команды представлено первым элементом командной строки, который находится после символа приглашения (нулевой аргумент). Оболочка обычно забирает первые символы

командной строки до первого пустого места (символа табуляции или пробела), а затем ищет команду с таким именем. Имя команды (первый аргумент) может быть указано в командной строке либо в виде простого имени файла, либо в виде полного имени. Например, можно вызвать команду ls любым из двух способов:

```
$ ls
$ /bin/ls
```

КЛЮЧ --HELP

При вызове с ключом --help многие утилиты выводят вспомогательное сообщение (иногда слишком детальное). Все утилиты, разработанные в рамках проекта GNU (см. историческую справку в главе 1), поддерживают этот ключ. Посмотрите следующий пример:

```
$ bzip2 --help
bzip2, a block-sorting file compressor.  Version 1.0.5, 10-Dec-2007.
   usage: bunzip2 [flags and input files in any order]
   -h --help           print this message
   -d --decompress     force decompression
   -z --compress       force compression
   -k --keep           keep (don't delete) input files
   -f --force          overwrite existing output files
...
   If invoked as 'bzip2', default action is to compress.
       as 'bunzip2',  default action is to decompress.
       as 'bzcat', default action is to decompress to stdout.
...
```

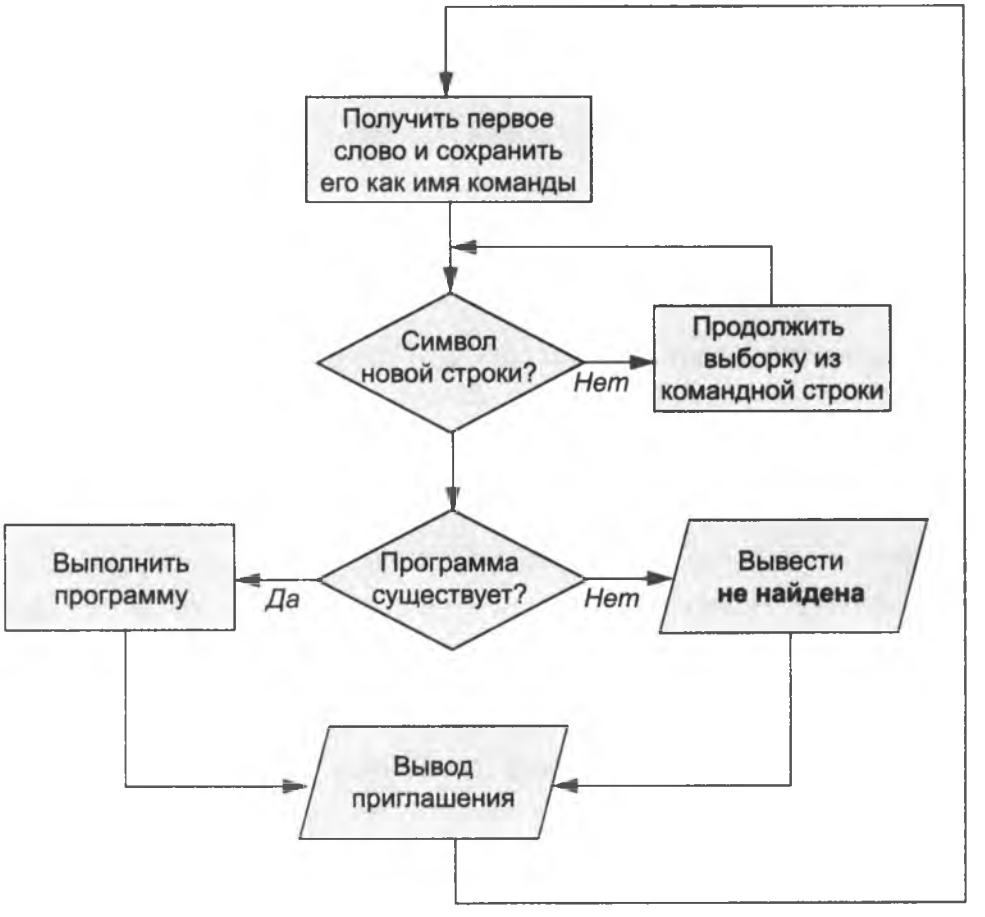

Рис. 5.1. Обработка командной строки

Оболочка не требует, чтобы имя команды появлялось в самом начале строки. Поэтому можно выстроить командную строку следующим образом:

```
$ >bb <aa cat
```

Эта команда запускает утилиту cat со стандартным входом, поступающим из файла по имени aa, и стандартным выходом, который перенаправляется в файл по имени bb. Когда оболочка встречает символы перенаправления (см. раздел «Перенаправление» на с. 146), она их распознает, обрабатывает их и имеющиеся у них аргументы перед поиском имени программы, вызываемой из командной строки. Это вполне допустимая по синтаксису, хотя и редко встречающаяся (и, возможно, несколько запутанная) командная строка.

Сравнение абсолютных и относительных путевых имен

Когда в командной строке дается абсолютное имя программы или относительное имя, не являющееся простым именем файла (то есть любое имя, включающее хотя бы один символ слэша), оболочка ведет поиск в указанном каталоге (в случае с командой /bin/ls – в каталоге /bin) файла с заданным именем, на запуск которого у вас есть права. Если задать простое имя файла программы, оболочка ищет в списке каталогов имя файла, соответствующее указанному имени, из числа тех имен файлов, на запуск которых у вас есть права. При этом оболочка просматривает не все каталоги, а только те, которые указаны в переменной PATH. Дополнительные сведения о переменной PATH можно найти в главах 8 и 9 (для оболочек bash и tcsh соответственно), в разделе «PATH: Указание тех мест, где оболочка ищет программы» на с. 312. Обратитесь также к описанию утилит which и whereis в главе 3.

Когда оболочка Bourne Again Shell (bash) не может найти исполняемый файл, она выводит следующее сообщение:

```
$ abc
bash: abc: command not found    Команда не найдена
```

Одна из причин, по которой оболочка может не отыскать файл, — отсутствие нужного каталога в вашей переменной PATH. При работе с оболочкой bash следующая команда может временно добавить к PATH рабочий каталог (.):

```
$ PATH=$PATH:.
```

Отсутствие желания включать свой рабочий каталог в PATH на постоянной основе может быть продиктовано соображениями безопасности (см. врезку ниже). Когда оболочка находит программу, но не может ее выполнить (например, из-за того, что у вас нет прав на выполнение нужного файла), она выводит примерно такое сообщение:

```
$ def
bash: ./def: Permission denied    В доступе отказано
```

Информацию об отображении прав доступа к файлу можно найти в разделе «ls –l: Отображение прав» на с. 115. Инструкции по изменению прав доступа изложены в разделе «chmod: Изменение прав доступа» на с. 116.

ПОПРОБУЙТЕ ВВЕСТИ КОМАНДУ В ВИДЕ ./КОМАНДА

Исполняемый файл в рабочем каталоге можно всегда запустить, если впереди имени его файла поставить символы «./». Например, если myprog является исполняемым файлом в рабочем каталоге, его можно запустить на выполнение следующей командой, независимо от установок, содержащихся в переменной PATH:

```
$ ./myprog
```

Рис. 5.2. Команда не знает, откуда пришел стандартный ввод или куда ушли стандартный вывод и стандартная ошибка

Выполнение командной строки

Процесс

Если оболочка найдет исполняемый файл с таким же именем, как у введенной команды, она запустит новый процесс. *Процесс* — это выполнение команды системой Linux (см. раздел «Процессы» главы 8). Оболочка открывает вызываемой программе доступ ко всем аргументам командной строки, включая ключи и имя самой команды. В ходе выполнения команды оболочка ждет завершения процесса и находится в неактивном состоянии, называемом *сном*. Когда программа завершит работу, она передает код возврата (см. раздел «$?: Статус выхода» на с. 458) оболочке. Оболочка возвращается в активный режим (просыпается), выводит приглашение и ждет следующей команды.

Оболочка не ведет обработку аргументов

Поскольку оболочка не обрабатывает аргументы командной строки, а просто передает их вызываемой программе, у нее нет возможности узнать, подходит ли тот или иной ключ (или другой аргумент) данной программе. Любые сообщения об ошибке или о порядке использования команды поступают из самой программы. Некоторые утилиты просто игнорируют неправильно указанные ключи.

Редактирование командной строки

Можно повторять и редактировать предыдущие команды и редактировать текущую командную строку. Дополнительные сведения содержатся в разделе «Повторение и редактирование командных строк» на с. 50, а также в посвященной оболочке bash главе 8, в разделе «Повторное выполнение и редактирование команд» на с. 326 и в посвященной оболочке tcsh главе 9, в разделе «Редактирование командной строки» на с. 381.

Стандартный ввод и стандартный вывод

Стандартный вывод — это место, куда программа может отправить информацию, например текст. Программа никогда не «знает», куда уходит информация, отправленная на стандартный вывод (рис. 5.2). Информация может попасть на принтер, в обычный файл или на экран. В следующих разделах показывается, что по умолчанию оболочка направляет стандартный вывод из команды на экран[1], и объясняется, как можно заставить оболочку перенаправить этот вывод в обычный файл.

[1] В данной книге термин *экран* используется для ссылки на экран, на окно эмулятора терминала или рабочей станции — другими словами, для ссылки на устройство, в котором оболочка отображает свое приглашение и выводит свои сообщения.

Стандартный ввод — это место, откуда программа получает информацию. Так же, как и со стандартным выводом, программа никогда не «знает», откуда поступает информация. В следующих разделах объясняется, как перенаправить стандартный ввод в команду таким образом, чтобы он поступал из обычного файла, а не с клавиатуры (откуда он поступает по умолчанию).

В дополнение к стандартному вводу и стандартному выводу, у работающей программы обычно есть место для отправки сообщений об ошибках: *стандартная ошибка*. Дополнительные сведения о работе со стандартной ошибкой содержатся в главе 8 (bash), в разделе «Перенаправление стандартной ошибки» на с. 288, и в разделе с таким же названием в главе 9 (tcsh).

По соглашению, процесс ожидает, что программа, которая его вызвала (зачастую это оболочка), установила стандартный ввод, стандартный вывод и стандартную ошибку, чтобы процесс мог незамедлительно воспользоваться ими. Вызываемый процесс не должен знать, какие файлы или устройства подключены к стандартному вводу, стандартному выводу или стандартной ошибке.

Но процесс может запросить ядро, чтобы получить информацию о том, к какому устройству подключен стандартный ввод, стандартный вывод или стандартная ошибка. Например, утилита ls отображает вывод в нескольких столбцах при выводе на экран, но создает на выходе только один столбец, если вывод перенаправлен в файл или в другую программу. Для определения того, что вывод пойдет на экран (a tty), утилита ls использует системный вызов isatty(). Кроме того, утилита ls может воспользоваться другим системным вызовом для определения ширины экрана, на который она отправляет свой вывод. После этого она изменяет свой вывод, чтобы он уместился в ширину экрана. Сравните вывод утилиты ls как таковой, и ее же вывод, когда вы отправляете его через канал утилите less.

Экран в качестве файла

В главе 4 были описаны обычные файлы, файлы каталогов, а также жесткие и мягкие ссылки. В Linux и в OS X имеется дополнительный тип файла: *файл устройства*. Этот файл находится в файловой структуре Linux, обычно в каталоге /dev, и представляет внешнее устройство, например экран и клавиатуру, принтер или дисковый привод.

Имя устройства, которое утилита who выводит после вашего имени пользователя — это имя файла экрана и клавиатуры. Например, когда утилита who отображает имя устройства pts/4, то полное имя экрана и клавиатуры — /dev/pts/4. В графической среде при работе с несколькими окнами каждое окно имеет свое собственное имя устройства. Для отображения имени устройства, с которого была подана команда, можно также воспользоваться утилитой tty. Хотя обычно в этом не возникает надобности, из этого файла можно читать данные или записывать их в него, как будто вы имеете дело с обычным текстовым файлом. При записи в него он отображает на экране все, что вы записали, а при просмотре происходит чтение того, что было введено с помощью клавиатуры.

Клавиатура и экран как стандартный ввод и стандартный вывод

При первоначальном входе в систему, оболочка направляет стандартный вывод ваших команд в файл устройства, который представляет экран (рис. 5.3). Такое направление вывода приводит к тому, что он появляется на экране. Оболочка направляет стан-

дартный ввод таким образом, чтобы он поступал из того же файла, чтобы ваши команды получали в качестве ввода все, что вы набираете на клавиатуре.

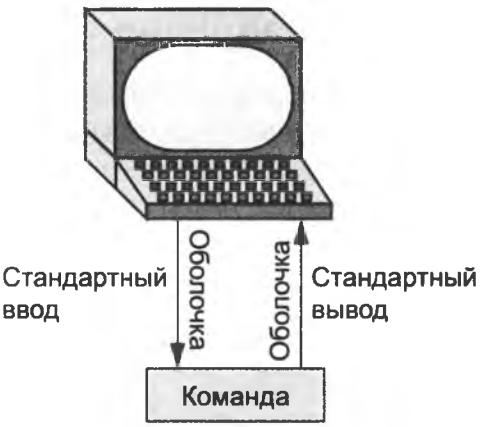

Рис. 5.3. По умолчанию стандартный ввод поступает с клавиатуры, а стандартный вывод отправляется на экран

Утилита cat

Утилита cat может послужить хорошим примером того способа, который положен в основу функционирования клавиатуры и экрана в качестве стандартных устройств ввода и вывода. При использовании утилиты cat она копирует файл на стандартный выход. Поскольку оболочка направляет стандартный выход на экран, cat отображает файл на экране.

До сих пор утилита cat получала входящую информацию из файла, чье имя (в виде аргумента) указывалось вами в командной строке. Когда этой утилите не передается никаких аргументов (то есть после ввода команды cat сразу же нажимается клавиша ВВОД), она получает входящую информацию от стандартного устройства ввода. Таким образом, при вызове без аргументов утилита cat построчно копирует стандартный ввод на стандартный вывод.

Чтобы посмотреть, как работает утилита cat, в ответ на приглашение оболочки наберите cat и нажмите клавишу ВВОД. Прямо под той строкой, которую вы вводите, появляется точно такая же строка. Это работает утилита cat. Поскольку оболочка связывает стандартный ввод утилиты cat с клавиатурой, а ее стандартный вывод — с экраном, когда вы набираете строку текста, то утилита cat копирует текст из стандартного ввода (клавиатуры) в стандартный вывод (на экран). Этот обмен данными показан в листинге 5.2.

Листинг 5.2. Утилита utility копирует стандартный ввод на стандартный вывод

```
$ cat
Это строка текста.
Это строка текста.
Cat продолжает копировать строки текста
Cat продолжает копировать строки текста
До тех пор, пока вы не нажмете CONTROL+D в начале
До тех пор, пока вы не нажмете CONTROL+D в начале
строки.
строки.
CONTROL+D
$
```

Комбинация CONTROL+D выдает сигнал EOF

Утилита cat продолжает копирование текста, пока вы не введете в строку символ, генерируемый комбинацией клавиш CONTROL+D. Эта комбинация отправляет сигнал EOF (end of file — конец файла) утилите cat, чтобы показать, что она достигла конца стандартного ввода и текста для копирования больше нет. Затем утилита cat завершает свою работу и возвращает управление оболочке, которая отображает приглашение.

Перенаправление

Термин *перенаправление* включает в себя различные способы, которыми можно заставить оболочку изменить стандартные ввод и вывод данных. По умолчанию оболочка связывает стандартный ввод и стандартный вывод команды с клавиатурой и экраном. Ее можно заставить перенаправить стандартный ввод или стандартный вывод любой команды, связав ввод или вывод с произвольной командой или файлом. В этом разделе показано, как перенаправить ввод в команду и вывод из нее на обычные файлы.

Перенаправление стандартного вывода

Символ перенаправления вывода (>) заставляет оболочку перенаправить вывод команды в указанный файл, а не на экран (рис. 5.4). Формат командной строки, перенаправляющий вывод, имеет следующий вид:

```
команда [аргументы] > имя_файла
```

где *команда* — это любая выполняемая программа (например, приложение или утилита), *аргументы* — это необязательные аргументы команды, а *имя_файла* — имя обычного файла, в который оболочка перенаправляет вывод.

На рис. 5.5 для демонстрации перенаправления используется команда `cat`. Этот рисунок отличается от рис. 5.4, где стандартный ввод *и* стандартный вывод связаны с клавиатурой и экраном. Ввод на рис. 5.4 по-прежнему поступает с клавиатуры. Но символ перенаправления вывода, использованный в командной строке, заставляет оболочку связать стандартный вывод команды `cat` с файлом sample.txt, указанным в командной строке.

После выдачи команды и набора текста, показанного на рис. 5.4, файл sample.txt содержит введенный текст. Команду `cat` можно использовать с аргументом `sample.txt` для отображения этого файла. В следующем разделе показан еще один способ использования `cat` для отображения файла.

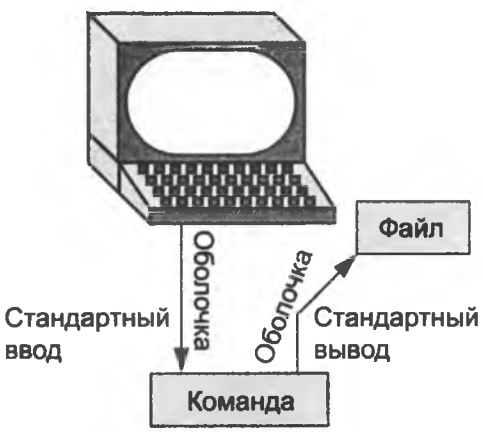

Рис. 5.4. Перенаправление стандартного вывода

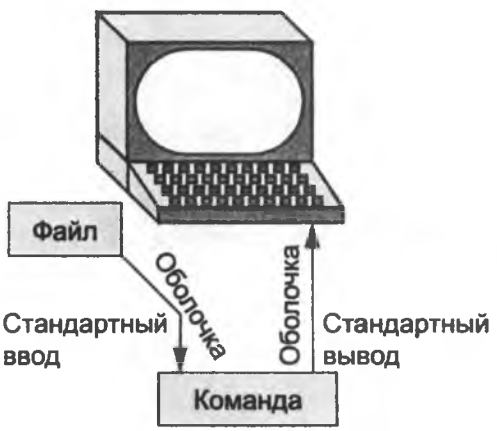

Рис. 5.5. Перенаправление стандартного ввода

В листинге 5.3 показано, что перенаправление стандартного вывода из команды `cat` является удобным способом создания файла без использования редактора. Недостаток этого способа состоит в том, что после набора текста и нажатия клавиши ВВОД вы уже не сможете редактировать текст. При вводе строки для удаления текста можно вос-

пользоваться клавишами забоя и удаления. Данная процедура удобна для создания небольших простых файлов.

Листинг 5.3. Команда cat с перенаправленным выводом

```
$ cat > sample.txt
Этот текст был введен с клавиатуры,
и cat скопировала его в файл.
Нажмите CONTROL+D, чтобы дать сигнал о конце файла.
CONTROL+D
$
```

ПЕРЕНАПРАВЛЕНИЕ ВЫВОДА МОЖЕТ УНИЧТОЖИТЬ ФАЙЛ – I

Будьте осмотрительны при перенаправлении вывода в файл. Если файл уже существует, оболочка его перепишет, уничтожив прежнее содержимое. Дополнительные сведения даны в совете «Перенаправление вывода может уничтожить файл — II» на с. 149.

В листинге 5.4 показано использование команды cat и символа перенаправления вывода для *объединения в цепочку* (catenate), то есть последовательного объединения нескольких файлов в один большой файл, от чего, собственно, и произошло имя утилиты cat. Первые три команды отображают содержимое трех файлов: stationery, tape и pens. Следующей командой является cat с тремя аргументами. При ее вызове с более чем одним аргументом команда cat поочередно копирует файлы в стандартное устройство вывода. Эта команда перенаправляет стандартный вывод в файл supply_orders. Завершающая команда cat показывает, что в файле supply_orders хранится содержимое трех исходных файлов.

Листинг 5.4. Использование команды cat для объединения файлов

```
$ cat stationery
2,000 sheets letterhead ordered:    10/7/08

$ cat tape
1 box masking tape ordered:         10/14/08
5 boxes filament tape ordered:      10/28/08

$ cat pens
12 doz. black pens ordered:         10/4/08

$ cat stationery tape pens > supply_orders

$ cat supply_orders
2,000 sheets letterhead ordered:    10/7/08
1 box masking tape ordered:         10/14/08
5 boxes filament tape ordered:      10/28/08
12 doz. black pens ordered:         10/4/08
$
```

Перенаправление стандартного ввода

Можно перенаправить не только стандартный вывод, но и стандартный ввод. *Символ перенаправления ввода* (<) заставляет оболочку перенаправить ввод в команду таким образом, чтобы его источником была не клавиатура, а указанный файл (см. рис. 5.5). Формат командной строки для перенаправления ввода имеет следующий вид:

команда [аргументы] < имя_файла

где *команда* — это любая выполняемая программа (например, приложение или утилита), *аргументы* — это необязательные аргументы команды, а *имя_файла* — имя обычного файла, на получение данных из которого оболочка перенаправила ввод.

В листинге 5.5 показана команда cat, ввод в которую перенаправлен на получение данных из файла supply_orders, созданного командой, показанной в листинге 5.4, а вывод из которой поступает на экран. Когда данные обычного файла заканчиваются, система автоматически выдает сигнал конца файла — EOF.

Утилиты, воспринимающие ввод из файла или со стандартного устройства ввода

Выдача команды cat с перенаправленным вводом приводит к тому же результату, что и выдача этой же команды с указанием имени файла в качестве аргумента. Утилита cat относится к классу Linux-утилит, работающих в данном стиле. К этому же классу утилит относятся lpr, sort, grep и Perl. Эти утилиты сначала исследуют командную строку, в которой они вызваны. Если в командную строку включено имя файла, то утилита осуществляет ввод данных из указанного файла. Если же имя файла не указано, то утилита вводит данные со стандартного устройства ввода. Такой метод работы присущ именно утилите или программе, а не оболочке или операционной системе.

Листинг 5.5. Команда cat с перенаправленным вводом

```
$ cat < supply_orders
2,000 sheets letterhead ordered:    10/7/08
1 box masking tape ordered:         10/14/08
5 boxes filament tape ordered:      10/28/08
12 doz. black pens ordered:         10/4/08
```

noclobber: Предотвращение перезаписи файлов

Оболочка предоставляет пользователям свойство noclobber, предотвращающее перезапись файла при использовании перенаправления. При использовании оболочки bash это свойство можно включить, установив значение noclobber вводом команды set -o noclobber. Аналогичная команда с ключом +o отменяет установку свойства noclobber. При использовании оболочки tcsh для аналогичных нужд применяются команды set noclobber и unset noclobber. При установленном свойстве noclobber перенаправление вывода в уже существующий файл приводит к тому, что оболочка выведет сообщение об ошибке и не выполнит команду. В следующем примере создается файл, используя команду touch, устанавливается свойство noclobber, предпринимается попытка перенаправить вывод из команды echo в только что созданный файл, затем отменяется установка свойства noclobber и снова осуществляется перенаправление: Показано выполнение этого примера для обеих оболочек.

Bash

```
$ touch tmp
$ set -o noclobber
$ echo "hi there" > tmp
bash: tmp: cannot overwrite existing file
Перезапись существующего файла невозможна
$ set +o noclobber
$ echo "hi there" > tmp
$
```

Tcsh
```
tcsh $ touch tmp
tcsh $ set noclobber
tcsh $ echo "hi there" > tmp
tmp: File exists.  Файл уже существует
tcsh $ unset noclobber
tcsh $ echo "hi there" > tmp
$
```

ПЕРЕНАПРАВЛЕНИЕ ВЫВОДА МОЖЕТ УНИЧТОЖИТЬ ФАЙЛ — II

В зависимости от используемой оболочки и настройки окружения, команда, подобня нижеприведенной, может привести к нежелательным результатам в выдаче:

```
$ cat orange pear > orange
cat: orange: input file is output file
(файл ввода является также и файлом вывода)
```

Несмотря на то что команда cat вывела сообщение об ошибке, оболочка уничтожит содержимое существовавшего до этого файла orange. Новый файл orange будет иметь такое же содержимое, как и у файла pear, поскольку первое действие, предпринимаемое оболочкой при обнаружении символа перенаправления (>), направлено на удаление содержимого исходного файла orange.

Если нужно объединить два файла в один, следует воспользоваться командой cat для помещения двух файлов во временный файл, а затем воспользоваться командой mv для переименования временного файла:

```
$ cat orange pear > temp
$ mv temp orange
```

То, что происходит в следующем примере, может иметь еще более неприятные последствия. Пользователь хочет осуществить поиск слова «apple» в файлах a, b и c, перенаправив вывод из команды grep в файл a.output. К сожалению, пользователь ввел имя файла в виде «a output», пропустив точку в имени файла и поставив вместо нее пробел:

```
$ grep apple a b c > a output
grep: output: No such file or directory
(Такой файл или каталог отсутствует)
```

Оболочка без тени сомнения удалит содержимое файла «a», а затем вызовет команду grep. Сообщение об ошибке промелькнет лишь на мгновение, не позволяя заподозрить что-то неладное в ходе выполнения команды. Даже после того как вы заметите сообщение об ошибке, может пройти немало времени, пока обнаружится, что содержимое файла «a» было уничтожено при таком «поиске».

Действие noclobber можно отменить, поместив символ конвейера (в tcsh используется восклицательный знак) после символа перенаправления (>|). В следующем примере пользователь создает файл, перенаправляя вывод команды date. Затем пользователь устанавливает переключатель noclobber и перенаправляет вывод опять в тот же самый файл. Оболочка выводит сообщение об ошибке. Затем пользователь ставит символ конвейера после символа перенаправления, и оболочка опять позволяет пользователю перезаписать файл.

```
$ date > tmp2
$ set -o noclobber
$ date > tmp2
bash: tmp2: cannot overwrite existing file
            Перезапись существующего файла невозможна
$ date >| tmp2
$
```

Дополнительные сведения об использовании свойства noclobber при работе с оболочкой tcsh даются в разделе «Переменные оболочки, работающие как переключатели» на с. 395.

Добавление стандартного вывода в файл

Символ добавления вывода (>>) заставляет оболочку добавить новую информацию в конец файла, оставляя нетронутой существующую информацию. Этот символ предоставляет удобный способ объединения двух файлов в один. Следующая команда демонстрирует действие символа добавления вывода. Вторая команда выполняет объединение, рассмотренное в предыдущей врезке «Внимание»:

```
$ cat orange
Это orange
$ cat pear >> orange
$ cat orange
Это orange
Это pear
```

Первая команда отображает содержимое файла orange. Вторая команда добавляет содержимое файла pear к файлу orange. Заключительная команда cat отображает полученный результат.

НЕ ПОЛАГАЙТЕСЬ НА СВОЙСТВО NOCLOBBER

Добавление вывода проще двухшаговой процедуры, показанной в предыдущей врезке «ВНИМАНИЕ», но нужно быть осмотрительным, обязательно поставив в строку оба знака «больше». Если при отключенном свойстве noclobber неосмотрительно поставить только один такой знак, оболочка перезапишет файл orange. Даже при включенном свойстве noclobber лучше все же на случай ошибки хранить архивные копии тех файлов, с которыми вы работаете.

Несмотря на то что свойство noclobber защищает от перезаписи файла при использовании перенаправления, оно не предохраняет от перезаписи файла при использовании команд cp или mv. У этих утилит имеется ключ –i (interactive — интерактивный режим), помогающий удержать вас от совершения подобной ошибки. В этом режиме утилиты требуют подтверждения ваших намерений до попытки перезаписи файла. Дополнительные сведения даны во врезке «Утилита cp может уничтожить файл» на с. 70.

В следующем примере показано, как создавать файл, содержащий дату и время (вывод из команды date), за которым следует список пользователей, которые вошли в систему (вывод из команды who). В первой строке листинга 5.6 осуществляется перенаправление вывода из команды date в файл по имени whoson.

Затем команда cat отображает этот файл. В следующем примере вывод из команды who добавляется к файлу whoson. В завершение команда cat отображает файл, содержащий вывод обеих утилит.

Листинг 5.6. Перенаправление и добавление вывода

```
$ date > whoson
$ cat whoson
Fri Mar 27 14:31:18 PST 2009
$ who >> whoson
$ cat whoson
Fri Mar 27 14:31:18 PST 2009
sam        console      Mar 27 05:00(:0)
```

```
max      pts/4        Mar 27 12:23(:0.0)
max      pts/5        Mar 27 12:33(:0.0)
zach     pts/7        Mar 26 08:45 (bravo.example.com)
```

/dev/null: отправка данных в никуда

Устройство /dev/null является *приемником данных*, часто упоминаемым как *пустое устройство*. Можно перенаправить в /dev/null вывод, который не нужно сохранять или отображать, и этот вывод бесследно исчезнет:

```
$ echo "hi there" > /dev/null
$
```

При чтении из устройства /dev/null получается пустая строка. Ввод следующей команды cat приведет к усечению файла по имени messages до нулевой длины, но к сохранению прав собственности на этот файл и прав доступа к нему:

```
$ ls -l messages
-rw-r--r--   1 max pubs 25315 Oct 24 10:55 messages
$ cat /dev/null > messages
$ ls -l messages
-rw-r--r--   1 max pubs 0 Oct 24 11:02 messages
```

Каналы

Оболочка использует *канал* для подключения стандартного вывода одной команды к стандартному вводу другой команды. Канал (часто называемый *конвейером*) имеет тот же эффект, что и перенаправление стандартного вывода одной команды в файл с последующим использованием этого файла в качестве стандартного ввода в другую команду. Канал позволяет избавиться от отдельных команд и этого промежуточного файла. В качестве символа для канала используется вертикальная черта (|). Синтаксис командной строки, использующей канал, имеет следующий вид:

```
команда_a [аргументы] | команда_b [аргументы]
```

Здесь канал используется в одной строке для получения такого же результата, к которому привело бы использование трех командных строк:

```
команда_a [аргументы]> temp
команда_b [аргументы]< temp
rm temp
```

В этой последовательности команд первая строка перенаправляет стандартный вывод из *команды_a* в промежуточный файл по имени temp. Вторая строка перенаправляет стандартный ввод для *команды_b* таким образом, чтобы он был получен из файла temp. Наконец, последняя строка дает команду на удаление файла temp. Команда, использующая канал, не только проще набирается, но и работает эффективнее, поскольку не создает промежуточного файла.

Утилита tr

Канал можно использовать с любыми утилитами Linux, воспринимающими входную информацию как из файла в командной строке, так и со стандартного ввода. Каналы можно также использовать с утилитами, воспринимающими входную информацию только со стандартного ввода. Например, утилита tr

(translate — перевод, см. часть V) получает входную информацию только со стандартного ввода. В самом простом варианте использования утилита tr имеет формат

```
tr строка1 строка2
```

Утилита tr ищет символы, которые соответствовали бы одному из символов в *строке1*. Как только соответствие будет найдено, утилита переводит соответствующий символ в *строке1* в тот символ, который ему соответствует в *строке2*. (Первый символ в *строке1* переводится в первый символ в *строке2* и т. д.) Утилита tr отправляет свою выходную информацию на устройство стандартного вывода. В обоих следующих примерах утилита tr отображает содержимое файла abstract, имеющее буквы a, b и c, переведенным соответственно в A, B и C:

```
$ cat abstract | tr abc ABC
$ tr abc ABC < abstract
```

Утилита tr не меняет содержимого исходного файла, она не может этого сделать, поскольку «не знает», что именно является источником информации на ее входе.

Утилита lpr Утилита lpr (line printer — построчный принтер, см. часть V) воспринимает входную информацию как из файла, так и с устройства стандартного ввода. Когда в командной строке за именем lpt набирается имя файла, утилита помещает файл с таким именем в очередь на печать. Когда имя файла в командной строке не указывается, утилита lpr получает входную информацию с устройства стандартного ввода. Использование каналов позволяет перенаправлять входную информацию для утилиты lpr. Первый набор команд, приведенный в листинге 5.7, показывает, как можно использовать утилиты ls и lpr с промежуточным файлом (temp) для отправки на принтер списка файлов, находящихся в рабочем каталоге. Если файл temp уже существует, то первая команда перезапишет его содержимое. Во втором наборе команд для отправки на принтер используется канал (при этом использование файла temp исключается).

Листинг 5.7. Канал

```
$ ls > temp
$ lpr temp
$ rm temp
```

или

```
$ ls | lpr
```

Команды, приведенные в листинге 5.8, перенаправляют вывод из утилиты who в файл temp, а затем отображают содержимое этого файла в отсортированном порядке. Утилита sort (см. раздел «sort: Отображение содержимого файла в определенном порядке» на с. 74) получает входную информацию из файла, указанного в командной строке, или, когда файл не указан, из устройства стандартного ввода. Свою выходную информацию она отправляет на устройство стандартного вывода. Утилита sort, показанная в командной строке листинга 5.8, получает входную информацию с устройства стандартного ввода, которое перенаправлено (<) на получение информации из файла temp. В информации, которую утилита sort отправляет на экран, отображается список пользователей, отсортированный в алфавитном порядке.

Листинг 5.8. Использование временного файла для хранения промежуточных результатов

```
$ who > temp
$ sort < temp
max       pts/4        Mar 24 12:23
max       pts/5        Mar 24 12:33
sam       console      Mar 24 05:00
zach      pts/7        Mar 23 08:45
$ rm temp
```

Поскольку утилита sort может получать входную информацию из устройства стандартного ввода или из файла, чье имя указывается в командной строке, при удалении символа < из листинга 5.8 получится точно такой же результат.

В листинге 5.9 показано, как аналогичный результат может быть достигнут без создания файла temp. При использовании канала оболочка перенаправляет выходную информацию из утилиты who на вход утилиты sort. А утилита sort получает входную информацию из устройства стандартного ввода, поскольку за ее именем в командной строке не указано имя файла.

Когда система используется многими людьми, а вам нужно получить информацию только об одном из этих людей, можно отправить выходную информацию из утилиты who утилите grep (см. раздел «grep: Поиск строки» на с. 72), используя для этого канал. Утилита grep выводит строку (или строки), содержащую указанную вами строку, в данном примере указана строка sam:

```
$ who | grep 'sam'
sam       console      Mar 24 05:00
```

Еще один способ, позволяющий справиться с выходной информацией, слишком объемной, чтобы поместиться на одном экране (например, со списком файлов в переполненном каталоге), заключается в использовании канала для отправки выходной информации через утилиту less или утилиту more (см. раздел «less и more: Обеспечение поэкранного вывода текстового файла» на с. 68).

```
$ ls | less
```

Утилита less обеспечивает поэкранное отображение текста. Для просмотра следующего экрана нужно нажать клавишу ПРОБЕЛ, а для просмотра еще одной строки — клавишу ВВОД. Для получения справки следует нажать клавишу h, а для выхода из просмотра — клавишу q.

Листинг 5.9. Канал выполняет работу временного файла

```
$ who | sort
max       pts/4        Mar 24 12:23
max       pts/5        Mar 24 12:33
sam       console      Mar 24 05:00
zach      pts/7        Mar 23 08:45
```

Фильтры

Фильтр — это команда, обрабатывающая входной поток данных для создания выходного потока данных. В командной строке, включающей фильтр, используется канал, связывающий стандартный вывод какой-нибудь команды со стандартным вводом фильтра. Еще один канал связывает стандартный вывод фильтра со стандартным

вводом другой команды. В качестве фильтров могут использоваться далеко не все утилиты.

В следующем примере утилита sort используется в качестве фильтра, который получает стандартный ввод из стандартного вывода утилиты who и использует канал для перенаправления стандартного вывода на стандартный ввод утилиты lpr. Эта командная строка отправляет отсортированную выходную информацию из утилиты who на принтер:

```
$ who | sort | lpr
```

Предыдущий пример демонстрирует эффективность работы оболочки и универсальность утилит Linux. Три утилиты: who, sort и lpr, не были специально сконструированы для совместной работы, но все они однообразно используют стандартный ввод и стандартный вывод. Используя оболочку для обработки ввода и вывода, вы можете соединять стандартные утилиты в командной строке в единое целое для достижения желаемых результатов.

tee: Отправка выходной информации по двум направлениям

Утилита tee (см. часть V) копирует свой стандартный ввод как в файл, так и на стандартное устройство вывода. Эта утилита соответствует своему имени (tee — тройник): она берет единый поток входной информации и отправляет его по двум направлениям. В листинге 5.10 выходная информация утилиты who отправляется через канал на стандартный ввод утилиты tee. Утилита tee сохраняет копию стандартного ввода в файле по имени who.out, а также отправляет вторую копию на стандартный вывод. Стандартный вывод утилиты tee проходит через канал на стандартный ввод утилиты grep, которая отображает только те строки, в которых содержится подстрока sam. Чтобы утилита tee добавляла информацию в файл, а не перезаписывала ее в этом файле, следует воспользоваться ключом -a (append — добавить).

Листинг 5.10. Утилита tee копирует стандартный ввод в файл и на стандартный вывод

```
$ who | tee who.out | grep sam
sam      console      Mar 24 05:00
$ cat who.out
sam      console      Mar 24 05:00
max      pts/4        Mar 24 12:23
max      pts/5        Mar 24 12:33
zach     pts/7        Mar 23 08:45
```

Запуск команды в фоновом режиме

Активный режим

До сих пор все показанные команды запускались в активном режиме. При запуске команды в *активном режиме* оболочка, прежде чем вывести очередное приглашение на ввод команды и позволить вам продолжить с ней работу, ждет, пока команда завершит свою работу. Когда команда запускается в *фоновом режиме*, вам не нужно ждать завершения ее работы перед тем, как запустить еще одну команду.

Задания

Задание — это серия из одной или нескольких команд, которые могут быть связаны друг с другом каналами. В окне или на экране может быть только одно активное задание, но в фоновом режиме могут выполняться сразу несколько заданий. Запуская одновременно более одного задания, вы задействуете многозадачность — одно из наиболее важных свойств системы Linux. Запуск команды в фоновом режиме может быть обусловлен длительным временем ее работы или отсутствием необходимости наблюдать за тем, как она выполняется. Она не задействует экран, благодаря чему его можно использовать для другой работы. Разумеется, при использовании графического интерфейса пользователя (GUI) для запуска еще одного задания вы можете открыть еще одно окно.

Номер задания, PID-номер

Чтобы запустить задание в фоновом режиме, нужно в конце командной строки, непосредственно перед нажатием клавиши ВВОД, поставить знак амперсанда (&). Оболочка присвоит заданию номер в виде небольшого числа и отобразит этот номер в квадратных скобках. Сразу же за номером задания оболочка отобразит присвоенный операционной системой идентификационный номер процесса — PID (process identification number), который будет представлен числом, превышающим по значению номер задания. Каждый из этих номеров идентифицирует задание, запущенное в фоновом режиме. Затем оболочка отображает очередное приглашение, и вы можете вводить новую команду. Когда задание, выполняемое в фоновом режиме, будет завершено, оболочка выведет сообщение, в котором будет фигурировать как номер задания, так и командная строка, использовавшаяся для запуска команды.

В следующем примере показывается запуск команды в фоновом режиме. Эта команда отправляет выходную информацию утилиты ls через канал утилите lpr, которая отправляет ее на принтер.

```
$ ls -l | lpr &
[1] 22092
$
```

Строка с символами [1], следующая за командной строкой, показывает, что оболочка присвоила этому заданию номер 1. Число 22092 является PID-номером первой команды этого задания. (Оболочка TC Shell показывает PID-номера всех команд, входящих в задание.) Когда выполнение этого фонового задания завершится, оболочка выведет сообщение

```
[1]+ Done          ls -l | lpr
```

(Вместо ls -l, оболочка может вывести что-либо подобное ls --color=always -l. Это отличие обусловлено тем, что утилита ls была превращена в псевдоним (см. раздел «Псевдонимы») строки ls --color=always на с. 341.)

Перевод задания из активного в фоновый режим

Комбинация CONTROL+Z

Активное задание можно приостановить (то есть остановить без отмены задания), нажав клавишу приостановки, в роли которой обычно используется комбинация CONTROL+Z. После ее нажатия оболочка останавливает процесс и отключает стандартный ввод от клавиатуры. Приостановленное задание можно перевести в фоновый режим и перезапустить, используя команду bg, за которой следует номер задания. Если есть только одно приостановленное задание, то номер

задания указывать не нужно. Получать входную информацию с клавиатуры может только активное задание. Чтобы подключить клавиатуру к программе, запущенной в фоновом режиме, вы должны перевести ее в активный режим. Когда в фоновом режиме работает только одно задание, следует набрать команду fg без аргументов. Когда в фоновом режиме работает более одного задания, нужно набирать команду fg или знак процента (%), после чего ввести номер того задания, которое нужно перевести в активный режим. Оболочка отобразит команду, использовавшуюся для запуска задания (в следующем примере — это promptme), и вы сможете ввести любые входные данные, необходимые для продолжения работы программы:

```
bash $ fg 1
promptme
```

Чтобы не мешать работе, которая проводится в активном режиме (на экране), следует перенаправить выходную информацию заданий, запущенных в фоновом режиме. Дополнительные сведения по управлению заданием даны в разделе «Управление заданиями» на с. 299. Чтобы получить более подробную информацию о фоновых задачах, обратитесь к разделу «| и & разделяют команды, но не только» на с. 297.

kill: Прерывание фонового задания

Клавиша прерывания (обычно это комбинация CONTROL+C) не может прервать выполнение фонового процесса; для этой цели следует воспользоваться утилитой kill, см. часть V. За именем команды kill в командной строке следует указать либо PID-номер процесса, работу которого нужно прервать, либо знак процента, за которым следует номер задания.

Определение PID-номера процесса с помощью утилиты ps

Если вы забыли PID-номер, для его отображения можно воспользоваться утилитой ps (process status — статус процессов, см. часть V). В следующем примере в качестве фонового задания запускается команда tail -f outfile (ключ -f (follow — следить) заставляет утилиту tail следить за файлом outfile, и отображать новые строки по мере их записи в файл), используется утилита ps для отображения PID-номера процесса, после чего задание прерывается с помощью команды kill:

```
$ tail -f outfile &
[1] 18228
$ ps | grep tail
18228 pts/4    00:00:00 tail
$ kill 18228
[1]+  Terminated              tail -f outfile
$
```

Определение номера задания с помощью утилиты jobs

Если вы забыли номер задания, то для вывода списка номеров заданий можно воспользоваться командой jobs. Следующий пример похож на предыдущий, за исключением того, что в нем для идентификации прерываемого задания используется номер задания, а не PID-номер процесса. Иногда сообщение о том, что задание прекращено, не появляется до тех пор, пока вы не нажмете клавишу ВВОД после того, как эта же клавиша будет нажата для выполнения команды kill.

```
$ tail -f outfile &
[1] 18236
$ bigjob &
[2] 18237
$ jobs
[1]-  Running                 tail -f outfile &
[2]+  Running                 bigjob &
$ kill %1
$ RETURN
[1]-  Terminated              tail -f outfile
$
```

Генерация имени файла и расширение имени

Групповые символы, подстановка

Когда оболочке даются сокращенные имена файлов, содержащие специальные символы, которые также называются *метасимволами*, она может генерировать имена файлов, соответствующие именам существующих файлов. Эти специальные символы также называются *символами-джокерами*, поскольку они действуют наподобие джокеров из карточной колоды. Когда в аргументах командной строки появляется один из этих символов, оболочка расширяет этот аргумент в отсортированный список имен файлов и передает этот список программе, вызываемой командной строкой. Имена файлов, содержащие эти специальные символы, называются *нестрогими ссылками на файлы*, поскольку они не ссылаются на какой-нибудь один конкретный файл. Процесс, который оболочка выполняет с этими именами файлов, называется *подстановкой имени*.

Нестрогие ссылки на файлы позволяют быстро ссылаться на группу файлов с похожими именами, экономя ваши силы на наборе каждого из этих имен. Они помогают также найти файл, чье имя невозможно вспомнить целиком. Если нестрогой ссылке на файлы не найдутся соответствующие имена файлов, оболочка, как правило, передает программе ссылку без ее расширения, то есть специальные символы и все остальное.

Специальный символ «?»

Вопросительный знак (?) является специальным символом, который заставляет оболочку генерировать имена файлов. Он соответствует любому одиночному символу в имени существующего файла. В следующей команде этот специальный символ используется в аргументе для утилиты lpr:

```
$ lpr memo?
```

Оболочка расширяет аргумент memo? и генерирует список файлов, имеющихся в рабочем каталоге, имя которых состоит из строки memo, за которой стоит любой одиночный символ. Затем оболочка передает этот список утилите lpr. Сама утилита даже «не подозревает», что оболочка сгенерировала имена тех файлов, с которыми она была вызвана. Если нестрогой ссылке не будет соответствовать ни одно из имен файлов, оболочка передаст строку в том виде, в котором она присутствует в командной строке (memo?), утилите lpr или, если она на это настроена, передаст ей нулевую строку (см. описание свойства nullglob в табл. 8.13 на с. 349).

В следующем примере используется утилита ls, сначала для вывода имен всех файлов, имеющихся в рабочем каталоге, а затем для вывода имен файлов, которые соответствуют нестрогой ссылке на файлы memo?:

```
$ ls
mem    memo12  memo9  memomax    newmemo5
memo   memo5   memoa  memos
$ ls memo?
memo5  memo9   memoa  memos
```

Нестрогая ссылка на файлы memo? не соответствует именам mem, memo, memo12, memomax и newmemo5. Вопросительный знак можно также использовать в середине нестрогой ссылки на файлы:

```
$ ls
7may4report  may4report     mayqreport  may_report
may14report  may4report.79  mayreport   may.report
$ ls may?report
may.report  may4report  may_report  mayqreport
```

Чтобы попрактиковаться в генерации имен файлов, можно использовать команды echo и ls. Утилита echo выводит те аргументы, которые передает ей оболочка:

```
$ echo may?report
may.report may4report may_report mayqreport
```

Сначала оболочка расширяет нестрогую ссылку на файлы в список всех файлов, имеющихся в рабочем каталоге, которые соответствуют строке may?report. Затем она передает этот список утилите echo, как будто вы ввели список имен файлов в качестве аргументов для этой утилиты. После этого утилита echo выводит список имен файлов.

Вопросительный знак не соответствует лидирующей точке (той самой, которая является признаком скрытого имени файла, см. раздел «Скрытые имена файлов» на с. 102). Когда нужно добиться соответствия именам файлов, которые начинаются с символа точки, нужно включать эту точку в нестрогую ссылку на файлы в явном виде.

Специальный символ «*»

Звездочка (*) выполняет функцию, похожую на ту, которую выполняет вопросительный знак, но соответствует в имени файла любому количеству символов, включая нулевое. В следующем примере сначала выводится список всех файлов рабочего каталога, а затем показываются три команды, которые выводят все имена файлов, начинающиеся со строки memo, заканчиваются строкой mo и содержат строку sam:

```
$ ls
amemo  memo.0612   memosally     memsam      user.memo
mem    memoa       memosam.0620  sallymemo
memo   memorandum  memosam.keep  typescript
$ echo memo*
memo memo.0612 memoa memorandum memosally memosam.0620 memosam.keep
$ echo *mo
amemo memo sallymemo user.memo
$ echo *sam*
memosam.0620 memosam.keep memsam
```

Нестрогая ссылка на файлы memo* не соответствует именам файлов amemo, mem, sallymemo и user.memo. Как и «?», звездочка не соответствует лидирующей точке в имени файла.

Скрытые имена файлов

Ключ -a заставляет утилиту ls отобразить скрытые имена файлов. Команда echo * не выводит . (обозначение рабочего каталога), .. (обозначение родительского каталога, в котором находится рабочий каталог), .aaa или .profile. В отличие от нее команда echo .* выводит эти четыре имени:

```
$ ls
aaa        memo.sally  sally.0612  thurs
memo.0612  report      saturday
$ ls -a
.   .aaa      aaa         memo.sally  sally.0612  thurs
..  .profile  memo.0612   report      saturday
$ echo *
aaa memo.0612 memo.sally report sally.0612 saturday thurs
$ echo .*
. .. .aaa .profile
```

В следующем примере нестрогая ссылка .p* не соответствует именам memo.0612, private, reminder и report. Команда ls .* заставляет утилиту ls вывести имена .private и .profile в дополнение к каталогу . (рабочий каталог) и каталогу .. (родительский каталог по отношению к рабочему каталогу). При вызове с такими же аргументами утилита echo выводит имена файлов (включая каталоги) рабочего каталога, которые начинаются с точки (.), но без содержимого каталогов.

```
$ ls -a
.       .private   memo.0612  reminder
..      .profile   private    report
$ echo .p*
.private .profile
$ ls .*
.private .profile

.:
memo.0612  private  reminder   report

..:


.
$ echo .*
. .. .private .profile
```

Аккуратно придерживаясь соглашений о присвоении имен файлам, можно использовать нестрогие ссылки на файлы с немалыми удобствами. Например, если имена всех текстовых файлов заканчиваются расширением .txt, вы можете ссылаться на эту группу файлов с помощью строки *.txt. В следующей команде это соглашение используется для отправки всех текстовых файлов рабочего каталога на принтер. Знак амперсанда (&) заставляет утилиту lpr работать в фоновом режиме.

```
$ lpr *.txt &
```

Специальные символы [ ]

Пара квадратных скобок [ ], в которую заключен список символов, заставляет оболочку искать имена файлов, содержащие отдельные символы. Если строка memo? соответствует всем именам со строками memo, за которыми следует любой символ, то строка memo[17a] накладывает более строгие ограничения: она соответствует только именам файлов memo1, memo7 и memoa. Квадратные скобки определяют класс символов, который включает в себя все символы, которые в них заключены. (В проекте GNU это называется списком символов, а класс символов в GNU — несколько иное понятие.) Оболочка обрабатывает аргумент, который включает класс символов, подставляя по одному из каждого символа, принадлежащего этому классу вместо квадратных скобок и их содержимого. Затем оболочка передает список соответствующих имен файлов вызываемой ею программе.

Каждое определение класса символов может заменить в имени файла только один символ. Квадратные скобки и их содержимое похожи на вопросительный знак, который заменяется только на один элемент класса символов.

Первая из следующего перечня команд выводит список имен всех файлов рабочего каталога, которые начинаются на a, e, i, o или u. Вторая команда выводит содержимое файлов с именами page2.txt, page4.txt, page6.txt и page8.txt.

```
$ echo [aeiou]*
…
$ less page[2468].txt
…
```

Дефис внутри квадратных скобок определяет диапазон символов внутри определения класса. Например, строка [6-9] представляет класс символов [6789], строка [a-z] представляет все буквы нижнего регистра английского алфавита, а строка [a-zA-Z] представляет все буквы английского алфавита как нижнего, так и верхнего регистра.

Следующие командные строки показывают три способа распечатки файлов с именами part0, part1, part2, part3 и part5. Каждая из этих командных строк заставляет оболочку вызывать утилиту lpr с пятью именами файлов:

```
$ lpr part0 part1 part2 part3 part5
$ lpr part[01235]
$ lpr part[0-35]
```

Первая командная строка указывает все пять имен явным образом. Вторая и третья командные строки используют нестрогие ссылки на файлы, в которых используются определения классов символов. Оболочка расширяет аргумент второй командной строки, включая в него все файлы с именами, начинающимися на part и заканчивающиеся любым из символов, представленных в классе символов. Класс символов имеет явное определение 0, 1, 2, 3 и 5. Третья командная строка также использует определение класса символов, но определяет этот класс в виде диапазона 0-3 плюс 5.

Следующая командная строка выводит на печать 39 файлов, от part0 и до part38:

```
$ lpr part[0-9] part[12][0-9] part3[0-8]
```

Первая из следующих командных строк выводит список файлов рабочего каталога, чьи имена начинаются с букв от a до m. Вторая выводит список файлов, чьи имена заканчиваются буквами x, y или z.

```
$ echo [a-m]*
...
$ echo *[x-z]
...
```

Когда сразу за открывающей квадратной скобкой ([) следует восклицательный знак (!) или знак вставки (^), они определяют класс символов, в котором строка, заключенная в квадратные скобки, соответствует любому символу, не встречающемуся в квадратных скобках. То есть строка [^tsq]* соответствует любому имени файла, которое не начинается с символов t, s или q.

В следующем примере показано, что строка *[^ab] соответствует именам файлов, которые не оканчиваются буквами a или b, а строка [^b-d]* соответствует именам файлов, которые не начинаются с букв b, c или d.

```
$ ls
aa ab ac ad ba bb bc bd cc dd
$ ls *[^ab]
ac ad bc bd cc dd
$ ls [^b-d]*
aa ab ac ad
```

Можно заставить класс символов соответствовать знаку дефиса (–) или закрывающей квадратной скобке (]), если поставить их сразу же перед закрывающей квадратной скобкой.

В следующем примере показано, что утилита ls не может интерпретировать нестрогие ссылки на файлы. Сначала утилита ls вызывается с аргументом ?old. Оболочка преобразует строку ?old в список соответствующих имен файлов, сохраняет его, передает имена утилите ls. Вторая команда похожа на первую, за исключением того, что символ вопросительного знака (?) процитирован (см. раздел «Специальные символы» на с. 65).

Оболочка не распознает этот вопросительный знак в качестве специального символа, а передает его утилите ls. Та, в свою очередь, генерирует сообщение об ошибке, в котором говорится, что она не может найти файл с именем ?old (поскольку файла с таким именем нет).

```
$ ls ?old
hold
$ ls \?old
ls: ?old: No such file or directory  Такой файл или каталог отсутствует
```

Так же, как и большинство других утилит и программ, ls не может интерпретировать нестрогую ссылку на файлы, эта работа возлагается на оболочку.

ИМЕННО ОБОЛОЧКА ОБРАБАТЫВАЕТ НЕСТРОГИЕ ССЫЛКИ НА ФАЙЛЫ

Именно оболочка, а не программа, которую она запускает, занимается обработкой нестрогой ссылки на файлы. В примерах в этом разделе утилиты (ls, cat, echo, lpr) вообще не видят нестрогих ссылок. Оболочка «расширяет» эти ссылки и передает утилитам список обычных имен файлов.

Встроенные команды

Встроенная команда представляет собой утилиту (также называемую командой), которая встроена в оболочку. У каждой оболочки есть свой собственный набор встро-

енных команд. При запуске встроенной команды оболочка не инициирует создание нового процесса. Следовательно, встроенные команды не только запускаются намного быстрее, но и могут влиять на среду окружения текущей оболочки. Поскольку встроенные команды используются точно так же, как и утилиты, обычно вы даже не догадываетесь о том, встроена ли утилита в оболочку, или же она представляет собой вполне самостоятельную программу.

Утилита echo, к примеру, является ничем иным, как встроенной в оболочку командой. Оболочка всегда выполняет встроенные в нее команды перед тем, как предпринимает попытку найти команду или утилиту с таким же именем. Более подробно встроенные команды рассматриваются в в разделе «Встроенные команды» на с. 465, список команд, встроенных в оболочку bash, приводится в табл. 10.8 на с. 484, а такой же список для оболочки tcsh приводится в табл. 9.10 на с. 407.

Вывод списка команд, встроенных в оболочку bash

Чтобы вывести список команд, встроенных в оболочку bash, нужно ввести команду `info bash` и выбрать меню Shell Builtin Commands. Затем нужно выбрать пункты меню Bourne Shell Builtins и (или) Bash Builtins. Вопросы, касающиеся использования утилиты info, рассмотрены в главе 2. Info-страницы, посвященные оболочке bash, являются частью пакета bash-doc, и, если этот пакет не установлен, вы сможете просматривать только man-страницу (даже если воспользуетесь утилитой info).

Поскольку оболочка bash было создана в рамках проекта GNU, на info-странице дается более качественная информация, чем на man-странице. Если вы хотите прочитать материал, посвященный встроенным командам на man-странице, перейдите на man-страницу builtins.

Вывод списка команд, встроенных в оболочку tcsh

При работе с оболочкой tcsh, следует ввести команду man tcsh для отображения man-страницы, посвященной tcsh, а затем провести поиск с помощью следующей команды: `/Builtin commands$` (поиск строкового значения в конце строки).

Заключение

Оболочка является интерпретатором команд системы Linux. Она сканирует командную строку, проверяя соблюдение синтаксических норм, а также извлекая имя команды и каждый из аргументов. Первый аргумент считается аргументом номер один, второй — номером два и т. д. Имя самой команды считается нулевым аргументом. Для изменения характера работы команды многие программы используют ключи. Большинство утилит Linux идентифицируют ключи по одиночному или двойному дефису перед ключом.

При вводе команды оболочка пытается найти исполняемую программу с именем, соответствующем имени команды. Когда программа найдена, оболочка запускает ее на выполнение. Когда программа не найдена, оболочка сообщает, что она не может найти или выполнить программу. Если команда представлена простым именем файла, оболочка просматривает каталоги, заданные в переменной PATH, пытаясь найти эту команду.

При выполнении команды оболочка присваивает стандартному вводу команды один файл, а ее стандартному выводу — другой файл. По умолчанию оболочка заставляет стандартный ввод команды получать информацию с клавиатуры, а ее стандартный вывод — отправлять информацию на экран. Можно приказать оболочке перенаправить стандартный ввод или стандартный вывод команды на любой файл или устройство. Можно также подключить (с помощью канала) стандартный вывод одной команды к стандартному вводу другой команды. Фильтр представляет собой команду, которая считывает входную информацию со своего стандартного ввода, подключенного к стандартному выводу одной команды, и записывает свой стандартный вывод в стандартный ввод другой команды.

Когда команда запускается в активном режиме, то перед выводом очередного приглашения, позволяющего вам продолжить работу, оболочка ждет завершения работы этой команды. Когда в конце командной строки помещен знак амперсанда (&), оболочка выполняет такую команду в фоновом режиме и тут же выводит новое приглашение. Если вы хотите вводить другие команды в ответ на приглашение оболочки, то запускайте медленные команды в фоновом режиме. Встроенная в оболочку команда jobs выводит список приостановленных заданий, а также заданий, выполняемых в фоновом режиме. В этот список включается номер каждого задания.

Оболочка интерпретирует специальные символы в командной строке, чтобы сгенерировать имена файлов. Вопросительный знак (?) замещает любой одиночный символ, а знак звездочки (*) — любое число символов. Одиночный знак может быть также представлен классом символов, то есть списком, заключенным в квадратные скобки. Ссылка, использующая символы-джокеры (групповые символы), предназначенная для представления списка из одного или нескольких имен в краткой форме, называется нестрогой ссылкой на файлы.

Встроенная команда — это утилита, которая встроена в оболочку. Каждая оболочка имеет собственный набор встроенных команд. Когда оболочка запускает встроенную команду, она не порождает нового процесса. Следовательно, встроенная команда запускается быстрее, а также может оказывать влияние на среду, в которой работает текущая оболочка.

Утилиты и встроенные команды, представленные в данной главе

В табл. 5.1 приводится список утилит, описанных в данной главе.

Таблица 5.1. Новые утилиты

Утилита	Функция
Tr	Отображает одну строку символов на другую строку символов
Tee	Отправляет стандартный ввод как в файл, так и на стандартный вывод
Bg	Перемещает процесс в фоновый режим
Fg	Перемещает процесс в активный режим
Jobs	Выводит список приостановленных заданий и заданий, запущенных в фоновом режиме

Упражнения

1. Чем обычно занимается оболочка при выполнении команды? Что нужно делать, если для запуска следующей команды не хочется ждать завершения работы предыдущей?
2. Используйте утилиту sort в качестве фильтра, чтобы переписать следующую последовательность команд:

```
$ sort list > temp
$ lpr temp
$ rm temp
```

3. Что такое PID-номер? Зачем нужны эти номера при запуске процессов в фоновом режиме? Какая утилита выводит PID-номера запущенных вами команд?
4. Предположим, что следующие файлы находятся в рабочем каталоге:

```
$ ls
intro      notesb     ref2       section1   section3   section4b
notesa     ref1       ref3       section2   section4a  sentrev
```

Введите команду для каждого из следующих заданий, используя групповые символы для выражения имен файлов как можно меньшим количеством символов.

   1) Выведите список файлов, имена которых начинаются со строки `section`.
   2) Выведите список, состоящий только из имен section1, section2 и section3.
   3) Выведите только имя файла intro.
   4) Выведите имена файлов section1, section3, ref1 и ref3.

5. Обратитесь к части V или к man-страницам, чтобы определить, какая из команд будет способна на следующие действия:
   1) Вывести количество строк в стандартном вводе, в которых содержится *слово* `a` или `A`.
   2) Вывести только те имена файлов в рабочем каталоге, в которых содержится сочетание символов (шаблон) `$(`.
   3) Вывести имена файлов в рабочем каталоге в обратном алфавитном порядке.
   4) Отправить список файлов в рабочем каталоге на принтер, отсортировав его по размеру файлов.
6. Введите команду для:
   1) перенаправления стандартного вывода из команды sort в файл phone_list. Предположим, в качестве входного используется файл numbers;
   2) перевода всех случаев появления символов `[` и `{` в символ `(`, и всех случаев появления символов `]` и `}` в символ `)` в файле permdemos.c. (Подсказка: воспользуйтесь описанием утилиты tr в части V.)
   3) создания файла book, содержащего информацию из двух других файлов: part1 и part2.
7. Утилиты lpr и sort воспринимают ввод либо из файла, указанного в командной строке, либо из стандартного ввода.
   1) Назовите две другие утилиты, работающие аналогичным образом.
   2) Назовите утилиту, которая воспринимает ввод только из устройства стандартного ввода.
8. Приведите пример команды, которая использует утилиту grep:
   1) вместе с перенаправлением как ввода, так и вывода;
   2) только с перенаправлением ввода;
   3) только с перенаправлением вывода;
   4) с каналом.

   В каком из предыдущих случаев утилита grep используется в качестве фильтра?

9. Объясните причину появления следующего сообщения об ошибке. Какие имена файлов будут выведены следующей командой ls?

```
$ ls
abc  abd  abe  abf  abg  abh
$ rm abc ab*
rm: cannot remove 'abc': No such file or directory
rm: невозможно удалить 'abc': Такой файл или каталог отсутствует
```

Упражнения повышенной сложности

1. При использовании в команде символа перенаправления вывода (>) оболочка тут же (до выполнения команды) создает выходной файл. Продемонстрируйте истинность этого утверждения.
2. Экспериментируя с переменными оболочки, пользователь Max случайно удалил свою переменную PATH. Он решил, что ему не нужна переменная PATH. Рассмотрите проблемы, с которыми он может столкнуться, и объясните причины их возникновения. Как он может без особых усилий вернуть переменной PATH ее исходное значение?
3. Предположим, что вам предоставлено право записи в файл, но не предоставлено право его удаления.
   1) Введите команду, которая уничтожит содержимое файла, не прибегая к услугам текстового редактора.
   2) Объясните, как можно получить права на изменение содержимого того файла, который вы не можете удалить.
4. Если вы случайно создали файл с именем, содержащим невыводимые на печать символы, например управляющие символы, генерируемые с помощью клавиши CONTROL, то как можно удалить этот файл?
5. Почему переменная noclobber не защищает вас от перезаписывания существующего файла при использовании утилиты cp или утилиты mv?
6. Почему имена команд и файлов, как правило, не имеют встроенных пробелов?

   Каким образом можно создать имя файла, в котором содержится пробел? Как можно удалить такой файл? (Это упражнение на сообразительность, которое не следует решать практическим способом. Если хотите поэкспериментировать, создайте каталог для хранения только экспериментальных файлов и работайте только в этом каталоге.)
7. Создайте файл по имени answer и введите следующую команду:

```
$ > answers.0102 < answer cat
```

   Объясните, что делает эта команда и почему она это делает. Как выглядит более удобный и привычный способ записи такой команды?

ЧАСТЬ II

Редакторы

Глава 6. Редактор vim **168**

Глава 7. Редактор emacs **220**

Глава 6. Редактор vim

В начале этой главы приводятся исторические сведения и описывается редактор vi, довольно своеобразный, мощный, иногда в чем-то непонятный, интерактивный, визуально-ориентированный текстовый редактор. Затем в ней дается руководство по использованию редактора vim (улучшенного vi — его клона, поставляемого или доступного в большинстве дистрибутивов Linux), предназначенного для создания и редактирования файлов. Основная часть руководства и вся остальная часть главы посвящены редактору vi и его клонам. После руководства в главе излагаются подробные сведения о многих vim-командах и объясняется порядок использования параметров настройки vim в соответствии с вашими запросами. В конце приводится краткий справочник по vim-командам.

История

До создания редактора vi стандартным редактором системы UNIX был ed (имеющийся на большинстве Linux-систем), построчный редактор, при работе с которым просмотр редактируемого материала был затруднен. Затем появился редактор ex[1] — усовершенствованная версия ed. Наиболее заметным преимуществом редактора ex над редактором ed являлась возможность редактирования данных по всему пространству дисплея, что позволяло работать с полным экраном текста, а не только с одной строкой. При работе с ex перейти в полноэкранный режим работы можно было с помощью команды `vi` (Visual mode — визуальный, или наглядный режим). Полноэкранное редактирование приобрело настолько широкую популярность среди пользователей, что разработчики редактора ex, добавили в него возможность начинать работу сразу в полноэкранном режиме, не заставляя сначала запускать ex, а затем вводить в него команду `vi`. Соответственно, они дали этой версии программы имя vi. Точно так же, как при работе с ex, можно было вызвать визуальный режим, при работе с vi можно было вернуться назад к работе с ex. Начните с запуска ex, дайте ему команду `vi` для переключения в визуальный режим, и, находясь в этом режиме, дайте редактору команду `Q`, чтобы перейти к использованию ex. Ввод команды `quit` приводит к выходу из редактора ex.

Клоны редактора vi

Система Linux предлагает сразу несколько версий (клонов) редактора vi. Наиболее популярными из них являются редакторы elvis (elvis.the-little-red-haired-girl.org), nvi (реализация исходного редактора vi, www.bostic.com/vi), vile (invisible-island.net/vile/vile.html) и vim (www.vim.org). Кроме тех функций, которые предоставляются исходным редактором vi, каждый клон предлагает еще и удобные дополнительные функции.

Примеры, приводимые в данной книге, основаны на работе с редактором vim. Ряд дистрибутивов Linux поддерживает сразу несколько версий vim. Например, Red

---

[1] Обычно программа ex является ссылкой на программу vi, которая на некоторых системах является версией редактора vim.

Hat предоставляет /bin/vi, минимальную, компактную сборку редактора vim, которая быстрее загружается, но предлагает меньше функций, и /usr/bin/vim, полнофункциональную версию vim.

При использовании вместо vim одного из его клонов или при работе с самим редактором vi вы можете заметить некоторые отличия между их работой и тем, как работают примеры, приведенные в данной главе. Команды редактора vim совместимы почти со всеми командами vi, а сам редактор запускается на многих платформах, включая Windows, Macintosh, OS/2, UNIX и Linux. Дополнительные сведения можно получить на главной странице веб-сайта, посвященного vim (www.vim.org), там же даются очень полезные советы по работе с этим редактором.

На что не способен vim

Редактор vim не может форматировать текст. В отличие от сложных текстовых процессоров типа OpenOffice.org Writer, он не выравнивает края и не предоставляет функции форматирования информации. Точнее, vim относится к современным текстовым редакторам, предназначенным для написания кода (C, HTML, Java и т. д.), кратких заметок и заготовок для систем форматирования текста, например для groff или troff. Для форматирования текстового файла, созданного с помощью vim, можно воспользоваться утилитой fmt (см. часть V).

Особенности данной главы

Поскольку vim обладает обширными и довольно мощными функциональными возможностями, в данной главе дается описание лишь нескольких из них. Тем не менее если vim окажется для вас совершенно новой программой, то даже этот ограниченный набор команд может показаться избыточным. Редактор vim предоставляет разнообразные способы выполнения большинства задач редактирования. Его изучение лучше всего начинать с поднабора команд, выполняющих самые основные задачи. Затем, по мере освоения редактора, можно изучать другие команды, позволяющие редактировать файл быстрее и эффективнее. Следующий раздел руководства представляет основной, самый востребованный набор команд и функций редактора vim, позволяющий создавать и редактировать файл.

Руководство. Использование vim для создания и редактирования файла

В этом разделе объясняется, как приступить к работе с vim, ввести текст, переместить курсор, исправить текст, сохранить файл на диске и выйти из vim. В руководстве рассматриваются три режима работы vim и объясняется порядок переключения между ними.

ИЗНАЧАЛЬНО ФАЙЛЫ РУКОВОДСТВА И ПОМОЩИ — VIMTUTOR И VIM HELP, НЕ УСТАНОВЛЕНЫ

Чтобы запустить программу vimtutor и получить помощь, которая рассматривается в разделе «Получение справки», следует установить пакет vim-runtime или vim-enhanced.

Программа vimtutor

В дополнение к работе с этим руководством, у вас может появиться желание воспользоваться учебной программой, посвященной редактору vim, которая называется vimtutor. Чтобы запустить этот учебник, нужно в качестве команды ввести его имя.

Определение типа терминала

Поскольку редактор vim пользуется функциями, присущими тем или иным терминалам, ему нужно сообщить, каким типом терминала (или эмулятора терминала) вы пользуетесь. На любых системах, в особенности если вы работаете в эмуляторе терминала, тип вашего терминала устанавливается автоматически.

Начало работы с vim

Начните работу с vim вводом следующей команды, предназначенной для создания и редактирования файла по имени practice:

```
$ vim practice
```

После нажатия клавиши ВВОД редактор vim меняет командную строку на экран, внешний вид которого показан на рис. 6.1.

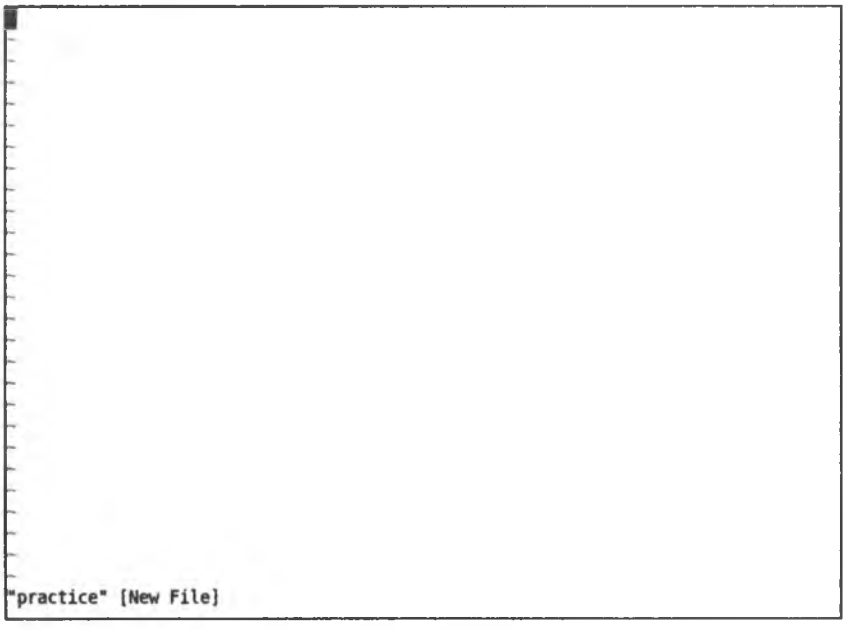

Рис. 6.1. Начало работы с vim

Тильды (~) в левой части экрана показывают, что файл пуст. Они исчезают по мере добавления к файлу текстовых строк. Если ваш экран имеет искаженный вид по сравнению с тем, который показан на рис. 6.1, то, вероятно, тип терминала установлен неверно.

Если запустить редактор vim на терминале, тип которого отсутствует в базе данных (terminfo), vim выведет сообщение об ошибке и переключится на тип терминала по умолчанию — ansi. Этот тип подходит ко многим терминалам. В следующем примере пользователь допустил опечатку при наборе vt100 и установил тип терминала vg100:

```
E558: Terminal entry not found in terminfo
    Запись о терминале в terminfo не найдена
'vg100' not known. Available builtin terminals are:
    'vg100' не известен. Доступны следующие встроенные драйверы терминалов:
    builtin_ansi
    builtin_xterm
    builtin_iris-ansi
```

```
    builtin_dumb
defaulting to 'ansi'
    по умолчанию установлен драйвер 'ansi'
```

РЕДАКТОР VIM МОЖНО ЗАПУСТИТЬ КОМАНДОЙ VI

На некоторых Linux-системах команда vi запускает редактор vim в режиме, совместимом с командами редактора vi (см. раздел «Параметр compatible» на с. 177).

Аварийный выход

Для переустановки типа терминала следует нажать клавишу **ESCAPE**, а затем ввести следующую команду для выхода из vim в приглашение оболочки:

```
:q!
```

При вводе двоеточия (:) vim перемещает курсор на последнюю строку экрана. Символы `q!` приказывают редактору vim завершить работу без сохранения всего, что вами сделано. (Обычно из vim так не выходят, поскольку, как правило, вам нужно сохранить свою работу, а не потерять ее.) После набора этой команды нужно нажать клавишу **ВВОД**. Как только оболочка выведет приглашение, нужно указать ей тип терминала и запустить vim еще раз.

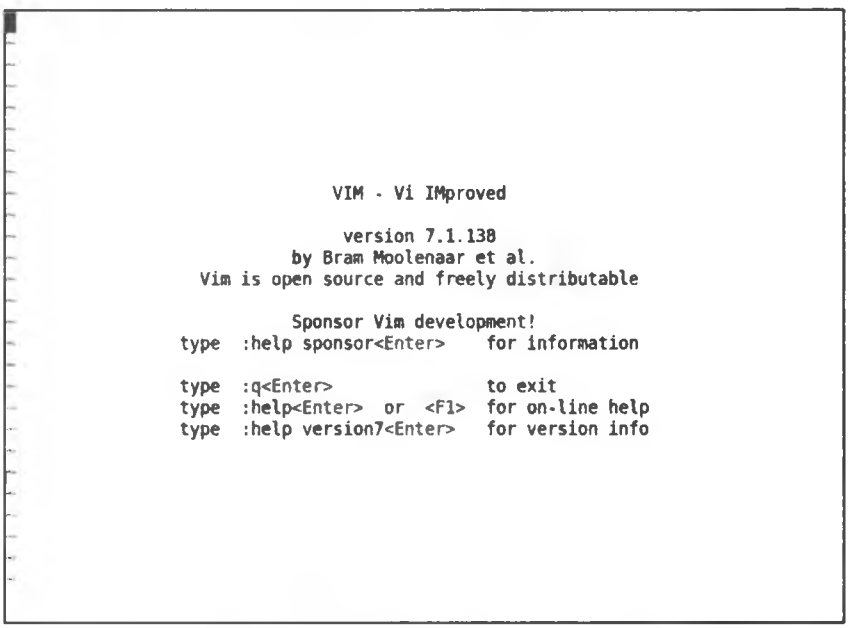

Рис. 6.2. Запуск vim без указания имени файла

Если вызвать vim без параметров командной строки (то есть без указания в командной строке имени файла), то vim решит, что вы новичок, и выведет подсказку, с чего можно начать работу (рис. 6.2).

Файл practice является новым, поэтому он не содержит никакого текста. В связи с этим vim выводит сообщение, аналогичное тому, которое показано на рис. 6.1 в строке состояния (последней строке) терминала, чтобы показать, что вами создан новый файл. При редактировании уже существующего файла vim показывает несколько первых строк этого файла и выводит в строку состояния информацию о состоянии этого файла.

Режимы ввода команд и текста

К двум рабочим режимам vim относятся *командный режим* (который также называется *Нормальным режимом*) и *режим ввода* (рис. 6.3). В командном режиме vim допускает ввод команд. Например, можно удалить текст или выйти из vim. Можно также скомандовать vim войти в режим ввода. В режиме ввода vim воспринимает все, что вы вводите как текст, который он отображает на экране. Чтобы вернуть vim в командный режим, следует нажать клавишу ESCAPE. По умолчанию редактор vim информирует вас, в каком режиме он находится: находясь в режиме ввода, он отображает в нижнем левом углу экрана слово INSERT (Вставка). Если в этом режиме vim не отображает слово INSERT, последуйте совету «Следите за режимом работы и за состоянием клавиши CAPS LOCK».

Следующая команда заставляет vim за набираемым текстом, который вы редактируете, отобразить номера строк:

```
:set number RETURN
```

Рис. 6.3. Режимы работы vim

Режим последней строки

Двоеточие (:), показанное в предыдущей команде, переводит vim в другой режим — *режим последней строки*. В этом режиме vim не выводит курсор за пределы последней строки экрана. При нажатии клавиши ВВОД с целью завершения ввода команды vim возвращает курсор на его прежнее место в тексте. Чтобы выключить нумерацию строк, следует ввести команду `:set nonumber` ВВОД.

Ввод текста

Команды i и a (режим ввода)

Чтобы после запуска редактора перейти к набору текста, нужно перевести vim в режим ввода. Для этого следует нажать клавишу i (дать команду вставки перед курсором) или клавишу a (дать команду добавления после курсора).

Если есть сомнения, что vim находится в режиме ввода, нажмите клавишу ESCAPE, и тогда vim вернется в командный режим, если он находился в режиме ввода, или подаст звуковой или визуальный сигнал, или вообще никак не среагирует, если вы уже находитесь в командном режиме. Нажав клавишу i или a еще раз, можно вернуть vim в режим ввода.

VIM ЧУВСТВИТЕЛЕН К РЕГИСТРУ БУКВ

При вводе команд в vim следует помнить, что этот редактор чувствителен к регистру букв. Иными словами, vim интерпретирует одну и ту же букву как две разные команды, в зависимости от того, в каком регистре вы вводите эту букву — в верхнем или нижнем. Следите за состоянием клавиши CAPS LOCK (SHIFTLOCK). Если, находясь в режиме ввода, активировать эту клавишу для ввода текста в верхнем регистре, а затем войти в командный режим, vim интерпретирует ваши команды как буквы в верхнем регистре. Такое поведение редактора может сбить с толку, поскольку vim будет выполнять совсем не те команды, которые вы хотели ввести.

Когда vim находится в режиме ввода, можно вводить текст, набирая его на клавиатуре. Если по мере ввода текст не появляется на экране, значит, редактор не находится в режиме ввода.

Чтобы продолжить обучение, введите простой текстовый абзац, показанный на рис. 6.4, нажимая клавишу ВВОД в конце каждой строки. Если не нажать ВВОД до того, как курсор достигнет правого края экрана или окна, vim автоматически перенесет текст, и он появится в начале следующей строки. В данной ситуации строки физически не соответствуют программным (логическим) строкам, поэтому редактировать их будет труднее. При работе с vim всегда можно исправить любую допущенную опечатку. Если вы заметили опечатку в уже введенной строке, то прежде чем продолжить работу, вы можете ее исправить (см. раздел «Правка текста в процессе его ввода» на с. 180), а остальные ошибки можно будет исправить чуть позже. Когда завершите ввод абзаца, нажмите клавишу ESCAPE, чтобы вернуться в командный режим.

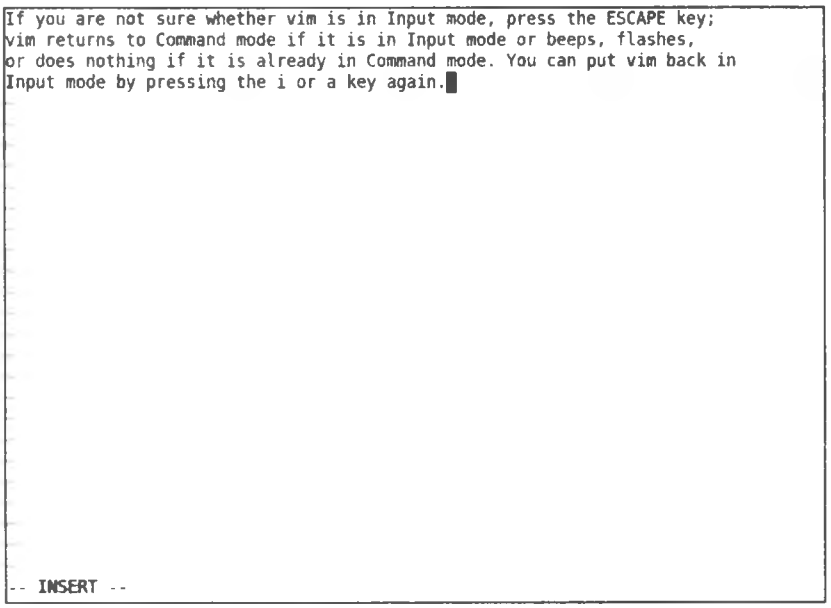

Рис. 6.4. Ввод текста в редакторе vim

Получение справки

Чтобы воспользоваться справочной системой vim, нужно установить пакет vim-runtime. Для получения помощи при работе с vim введите команду `:help [`*функция*`]`, после чего нажмите ВВОД (при вводе команды вы должны находиться в командном режиме). Двоеточие перемещает курсор на последнюю строку экрана. Если набрать команду `:help`, то vim выведет введение в справку редактора vim (рис. 6.5). Каждая область экрана, отображающая файл, как те две области, что показаны на рис. 6.5, является

«окном» редактора vim. Темная полоса в нижней части каждого vim-окна содержит имя файла, текст которого отображен выше этой полосы. На рис. 6.5 основную часть экрана занимает файл help.txt (верхнее vim-окно). Редактируемый файл (practice) занимает несколько строк в нижней части экрана (нижнее vim-окно).

Прочитайте введение в справку, пролистывая текст по мере чтения. Чтобы опуститься вниз по тексту на одну строку, нажимайте клавишу j или клавишу СТРЕЛКА ВНИЗ; чтобы прокрутить текст, перемещая курсор вниз или вверх сразу на половину размера окна, нажмите комбинацию CONTROL+D или CONTROL+U. Чтобы закрыть окно справки, введите команду `:q`.

```
help.txt*       For Vim version 7.1.  Last change: 2006 Nov 07

                      VIM - main help file
                                                                    k
     Move around:  Use the cursor keys, or "h" to go left,        h   l
                   "j" to go down, "k" to go up, "l" to go right.   j
Close this window: Use ":q<Enter>".
   Get out of Vim: Use ":qa!<Enter>" (careful, all changes are lost!).

Jump to a subject: Position the cursor on a tag (e.g. |bars|) and hit CTRL-].
   With the mouse: ":set mouse=a" to enable the mouse (in xterm or GUI).
                   Double-click the left mouse button on a tag, e.g. |bars|.
        Jump back: Type CTRL-T or CTRL-O (repeat to go further back).

Get specific help: It is possible to go directly to whatever you want help
                   on, by giving an argument to the |:help| command.
                   It is possible to further specify the context:
                                                        *help-context*
                        WHAT                  PREPEND    EXAMPLE
                    Normal mode command       (nothing)   :help x
help.txt [Help][RO]
If you are not sure whether vim is in Input mode, press the ESCAPE key;
vim returns to Command mode if it is in Input mode or beeps, flashes,
or does nothing if it is already in Command mode. You can put vim back in
Input mode by pressing the i or a key again.
~
practice
"help.txt" [readonly] 216L, 8010C
```

Рис. 6.5. Основной экран Справки (Help) редактора vim

Чтобы получить информацию по командам вставки, перейдите в командный режим и введите команду `:help insert` (рис. 6.6).

```
help.txt*       For Vim version 7.1.  Last change: 2006 Nov 07

                      VIM - main help file
                                                                    k
     Move around:  Use the cursor keys, or "h" to go left,        h   l
                   "j" to go down, "k" to go up, "l" to go right.   j
Close this window: Use ":q<Enter>".
   Get out of Vim: Use ":qa!<Enter>" (careful, all changes are lost!).

Jump to a subject: Position the cursor on a tag (e.g. |bars|) and hit CTRL-].
   With the mouse: ":set mouse=a" to enable the mouse (in xterm or GUI).
                   Double-click the left mouse button on a tag, e.g. |bars|.
        Jump back: Type CTRL-T or CTRL-O (repeat to go further back).

Get specific help: It is possible to go directly to whatever you want help
                   on, by giving an argument to the |:help| command.
                   It is possible to further specify the context:
                                                        *help-context*
                        WHAT                  PREPEND    EXAMPLE
                    Normal mode command       (nothing)   :help x
help.txt [Help][RO]
If you are not sure whether vim is in Input mode, press the ESCAPE key;
vim returns to Command mode if it is in Input mode or beeps, flashes,
or does nothing if it is already in Command mode. You can put vim back in
Input mode by pressing the i or a key again.
~
practice
"help.txt" [readonly] 216L, 8010C
```

Рис. 6.6. Справка по командам вставки (insert)

Исправление текста в процессе его ввода

Те клавиши, с помощью которых поддерживался ввод и корректировка данных в командной строке оболочки, выполняют такие же функции и в режиме ввода редактора vim. В их число входят клавиши стирания, удаления строки и слова (обычно это соответственно CONTROL+H, CONTROL+U и CONTROL+W). Хотя vim после перемещения курсора через этот текст может и не убрать удаленный текст с экрана, он удалит его по мере набора текста поверх ранее набранных символов или же после нажатия клавиши ВВОД.

Перемещение курсора

Чтобы удалять, вставлять или исправлять текст, нужно перемещать курсор по экрану. Когда vim находится в командном режиме, для перемещения курсора можно использовать клавиши ввода, пробела и стрелок. Если вы предпочитаете держать руки ближе к центру клавиатуры, а также если ваш терминал не имеет клавиш стрелок или если используемый вами эмулятор терминала не поддерживает этих клавиш, можно воспользоваться клавишами h, j, k и l (буква «l» в нижнем регистре) для перемещения курсора влево, вниз, вверх или вправо соответственно.

Удаление текста

Команда x (Удаление символа)
Команда dw (Удаление слова)
Команда dd (Удаление строки)

Одиночный символ можно удалить, если, перемещая курсор, закрыть им удаляемый символ, а затем ввести команду `x`. Слово можно удалить, если поставить курсор на первую букву этого слова, а затем ввести команду `dw` (Delete word — удалить слово). Строку текста можно удалить, если переместить курсор на любое место этой строки, а затем ввести команду `dd`.

Отмена действий для устранения ошибок

Команда u (Отмена)

Если вы по ошибке удалили символ, строку или слово или ввели любую команду, которую хотели бы отменить, введите команду `u` (Undo — Отменить) сразу же после ввода команды, которую вы хотите отменить. Редактор vim вернет текст в то состояние, в котором он был до ввода последней команды. Если ввести команду `u` еще раз, vim отменит команду, которая была введена непосредственно перед той командой, действия которой были только что отменены. Этот прием можно использовать для возвращения назад через многие свои действия. Но при установке параметра `compatible` (совместимость, см. раздел «Параметр compatible» на с. 177) vim может отменить только самые последние изменения.

Команда :redo (Вернуть отмененное действие)

Если отменена та команда, которую вы не собирались отменять, используйте команду «Вернуть отмененное действие» (Redo): нажмите комбинацию CONTROL+R или введите команду `:redo` (нажав после ее набора клавишу ВВОД). Редактор vim вернет действие отмененной команды. Как и при использовании команды «отменить» (Undo), вы можете вводить в строке команду Redo многократно. Но при установке параметра `compatible` (совместимость, см. раздел «Параметр compatible») vim может осуществить возврат только самых последних отмененных действий.

Ввод дополнительного текста

Команда i (Вставить)
Команда a (Добавить)

Когда нужно вставить новый текст внутри уже существующего текста, переместите курсор на тот символ, который будет следовать за вставляемым новым текстом. Затем введите команду `i` (Insert — Вставить), чтобы перевести vim в режим ввода, введите новый текст и нажмите клавишу ESCAPE для возвращения в командный режим. В качестве альтернативы можно установить курсор на позицию символа, который предшествует новому тексту, и воспользоваться командой `a` (Append — Добавить).

Команды o и O (Открыть)

Для ввода одной или нескольких строк установите курсор над той строкой, над которой нужно вставлять новый текст. Затем введите команду `o` (Open — Открыть). Редактор vim открывает пустую строку ниже той строки, на которую был установлен курсор, устанавливает курсор на новую строку и входит в режим ввода. Введите новый текст, нажимая в конце каждой строки клавишу ВВОД. Закончив ввод текста, нажмите клавишу ESCAPE, чтобы вернуться в командный режим. Команда `O` работает точно так же, как и команда `o`, но открывает пустую строку над той, на которой установлен курсор.

Правка текста

Для правки текста следует использовать команду `dd`, `dw` или `x`, чтобы удалить неверный текст, а после этого можно воспользоваться командой `i`, `a`, `o` или `O`, чтобы вставить правильный текст.

Например, для изменения слова `pressing` на слово `hitting` (см. рис. 6.4) можно воспользоваться клавишами стрелок, чтобы переместить курсор на букву `p` в слове `pressing`, а затем ввести команду `dw`, чтобы удалить слово `pressing`. После этого нужно перевести vim в режим ввода, для чего ввести команду `i`, набрать слово `hitting`, поставить за ним пробел и нажать клавишу ESCAPE. Слово будет изменено, и vim окажется в командном режиме в ожидании следующей команды. Сокращенным вариантом двух команд, команды `dw`, за которой следует команда `i`, является команда `cw` (Change word — Замена слова). Команда `cw` переводит `vim` в режим ввода.

РАЗРЫВ СТРАНИЦЫ ДЛЯ ПРИНТЕРА

Комбинация CONTROL+L генерирует символ, сообщающий принтеру, что он должен перейти к началу следующей страницы. Этот символ можно ввести в любом месте документа, нажав комбинацию CONTROL+L в режиме ввода. Если символ ^L не появится, то перед CONTROL+L следует нажать CONTROL+V.

Завершение сеанса редактирования

В процессе редактирования текста vim хранит его в области, которая называется *рабочим буфером*. Когда редактирование будет завершено, содержимое рабочего буфера следует записать в файл на диске, чтобы отредактированный текст был сохранен и доступен для дальнейшего использования.

Переведите vim в командный режим и воспользуйтесь командой `ZZ` (нужно использовать буквы `Z` в верхнем регистре), чтобы записать только что введенный текст на диск и завершить сеанс редактирования. После ввода команды `ZZ` vim возвращает

управление оболочке. Если ваша работа не требует сохранения, можно выйти из редактора с помощью команды `:q!`.

НЕ ПУТАЙТЕ ZZ С CONTROL+Z

При выходе из vim с помощью команды ZZ набирайте именно сочетание ZZ, а не комбинацию CONTROL+Z (которая обычно является клавишей приостановки). При нажатии CONTROL+Z vim исчезает с экрана, почти как при выходе из него. Однако фактически vim будет продолжать работу в фоновом режиме, и в нем будет находиться ваша несохраненная работа. Просмотрите раздел «Управление заданиями» на с. 299. Если вы попытаетесь запустить редактирование того же самого файла с помощью новой команды vim, редактор vim выведет сообщение о замене файла (рис. 6.7).

Параметр compatible

Параметр `compatible` (совместимость) придает редактору vim более высокий уровень совместимости с редактором vi. По умолчанию этот параметр не установлен. Если вы ранее не работали с vi, то можете проигнорировать этот параметр.

Установка параметра `compatible` изменяет многие аспекты работы редактора vim. Например, когда параметр `compatible` установлен, команда Отменить (Undo) способна отменить только самые последние изменения (см. раздел «Команды Отменить (u и U)» на с. 189). Для сравнения: когда параметр `compatible` не установлен, команду отмены можно вызывать для куда большего количества изменений. Когда параметр `compatible` влияет на работу команды, этот случай в данной главе оговаривается особо. Для получения более подробных сведений о параметре `compatible` следует ввести команду `:help compatible` **ВВОД**. Для отображения полного списка отличий редактора vim от исходного редактора vi следует воспользоваться командой `:help vi-diff` **ВВОД**. Команда `help` рассмотрена в разделе «Получение справки» на с. 173.

Чтобы установить параметр `compatible`, нужно в командной строке использовать ключ `-C`, а чтобы снять его установку, воспользуйтесь ключом `-N`. Информация по изменению установок параметра `compatible` в процессе работы редактора vim приводится в разделе «Установка параметров непосредственно из редактора vim» на с. 203.

Введение в функции редактора vim

В этом разделе рассматриваются интерактивная справка, режимы работы, рабочий буфер, аварийные процедуры и другие функции редактора vim. Чтобы понять, какие именно функции встроены в конкретную сборку, введите команду vim, поставив за ней ключ `--version`.

Интерактивная справка

Ранее уже кратко упоминалось, что vim при работе с ним содержит справку. Чтобы вывести информацию о той или иной функции, нужно ввести команду `:help` *функция*. По мере прокрутки различных текстов справки вам будут встречаться слова, по обе стороны от которых будет стоять по вертикальной черте, например `|tutor|`. Эти слова — *активные ссылки*: переместите курсор на активную ссылку и нажмите ком-

бинацию CONTROL+], чтобы перейти на текст по этой ссылке. Чтобы вернуться на прежнее место в тексте справки, воспользуйтесь комбинацией CONTROL+o (с буквой «o» в нижнем регистре). Слова активных ссылок можно также использовать вместо *функций*. Например, вы можете увидеть ссылку `|credits|` и ввести команду `:help credits` ВВОД, чтобы получить дополнительную информацию по теме `credits`. Чтобы закрыть окно справки, следует воспользоваться командой `:q!`.

К наиболее востребованным для изучения с помощью справочной системы *функциям* обычно относятся `insert` (вставить), `delete` (удалить) и `opening-window` (открыть окно). Хотя востребованность последней функции не столь очевидна, осмысление имен функций будет приходить по мере накопления опыта работы с vim. Для просмотра полного списка файлов справки можно ввести команду `:help doc-file-list`.

Хотя vim является бесплатной, автор просит, чтобы вы пожертвовали деньги, которые вы готовы потратить на подобную программу, в помощь детям Уганды (дополнительные сведения об этом выводятся по команде `:help iccf`).

Терминология

В данной главе используются следующие термины:

Текущий символ — Символ, на котором находится курсор.

Текущая строка — Строка, на которой находится курсор.

Строка состояния — Последняя (самая нижняя) строка экрана. Эта строка зарезервирована для режима последней строки и информации о состоянии. Редактируемый текст в этой строке не появляется.

Режимы работы

Редактор vim является частью редактора ex и имеет пять режимов работы:

- Командный режим редактора ex.
- Режим ввода редактора ex.
- Командный режим редактора vim.
- Режим ввода редактора vim.
- Режим последней строки редактора vim.

Находясь в командном режиме, vim воспринимает введенный текст в качестве команд, реагируя на каждую команду после ее ввода. Символы, набираемые в данном режиме, им не отображаются. А находясь в режиме ввода, vim воспринимает и отображает введенные символы как текст, который, в конечном итоге, помещает в редактируемый вами файл. Все команды, начинающиеся с двоеточия (`:`), переводят vim в режим последней строки. Двоеточие заставляет курсор переместиться на экране в строку состояния, в которой вы вводите остальную часть команды.

Кроме позиции курсора, есть еще одна существенная разница между режимом последней строки и командным режимом. При вводе команды в командном режиме вы не завершаете команду нажатием клавиши ВВОД. В отличие от этого все команды, введенные в режиме последней строки, должны завершаться нажатием клавиши ВВОД.

Обычно режимы редактора ex не используются. Когда в данной главе даются ссылки на режим ввода и командный режим, имеются в виду режимы редактора vim, а не режимы редактора ex.

В начале сеанса редактирования редактор vim находится в командном режиме. Ряд команд, включая Вставить (Insert) и Добавить (Append), переводят vim в режим редактирования. При нажатии клавиши ESCAPE vim всегда возвращается в командный режим.

Команды Изменить (Change) и Заменить (Replace) сочетают в своей работе командный режим и режим ввода. Команда Изменить (Change) удаляет текст, который нужно изменить, и переводит vim в режим ввода, чтобы вы могли вставить новый текст. Команда Заменить (Replace) удаляет переписываемые символ(ы) и вставляет новый, введенный вами символ (или символы). На рис. 6.3 показаны режимы и методы для переходов между ними.

СЛЕДИТЕ ЗА РЕЖИМОМ РАБОТЫ И ЗА СОСТОЯНИЕМ КЛАВИШИ CAPS LOCK

Почти все, что вводится вами в командную строку, имеет для vim какое-нибудь значение. Если вы полагаете, что vim находится в режиме ввода, когда он на самом деле находится в командном режиме, ввод текста может привести к непредвиденным результатам. При практическом изучении vim проверьте включение параметра showmode (параметр отображения режима работы, включаемый по умолчанию (табл. 6.7), чтобы понять, какой режим вами используется. Также принести пользу может включение строки состояния путем ввода команды :set laststatus=2 (см. табл. 6.7).

Также обращайте внимание на состояние клавиши CAPS LOCK. В командном режиме набор букв верхнего регистра выдает иные результаты, чем набор этих же букв в нижнем регистре. Когда в ответ на какую-нибудь команду vim совершит «неверное» действие, можно попасть в непонятную ситуацию.

Отображение информации

Чтобы информировать о всем происходящем в процессе сеанса редактирования, редактор vim использует строку состояния и несколько символов.

Строка состояния

Редактор vim выводит информацию о состоянии в самую нижнюю строку области отображения. Эта информация включает сообщения об ошибках, сведения об удалении или добавлении блоков текста и сведения о состоянии файла. Кроме этого vim отображает в строке состояния команды, вводимые в режиме последней строки.

Обновление экрана

Иногда изображение на экране может искажаться или выводится внакладку. Когда vim помещает символы на экран, то иногда вместо удаления строки на нем остается символ @. Когда информация, выводимая из программы, перемешивается с отображением рабочего буфера, возникает дополнительная неразбериха. Выведенная информация *перестает* быть частью рабочего буфера, и все, что делается с ней, воздействует только на отображение на экране. Если информация на экране пошла внакладку, нажмите клавишу ESCAPE, чтобы перевести vim в командный режим, и нажмите комбинацию CONTROL+L, чтобы перерисовать (обновить) экран.

Знак тильды (~)

Если на экране отображается конец файла, vim помечает строки, которые могут появиться после конца файла знаком тильды (~) в левой части экрана. Когда начинается редактирование нового файла, редактор vim помечает этим символом каждую строку экрана (за исключением первой).

Правка текста в процессе его ввода

Когда vim находится в режиме ввода, для стирания и последующей правки текста можно пользоваться клавишами стирания и удаления строки. Для стирания слов можно также пользоваться комбинацией CONTROL+W.

Рабочий буфер

Редактор vim всю свою работу выполняет в рабочем буфере. В начале сеанса редактирования vim считывает редактируемый файл с диска в рабочий буфер. Во время сеанса редактирования он производит все изменения в этой копии файла, но не вносит их в файл на диске до тех пор, пока вы не запишите содержимое рабочего буфера обратно на диск. Обычно, завершая сеанс редактирования, вы заставляете vim записать содержимое рабочего буфера, что приводит к окончательному внесению изменений в текст. При редактировании нового файла vim создает файл при записи содержимого рабочего буфера на диск, обычно это происходит при завершении сеанса редактирования.

Хранение редактируемого текста в рабочем буфере имеет свои преимущества и недостатки. Если вы случайно завершите сеанс редактирования без записи содержимого рабочего буфера, вся ваша работа пропадет. Но, если вы непреднамеренно внесете какие-то масштабные изменения (например, удалите все содержимое рабочего буфера), можно будет завершить сеанс редактирования без внесения изменений.

Чтобы просмотреть файл в процессе работы с ним в редакторе vim, не внося при этом в него изменений, можно воспользоваться утилитой view:

```
$ view filename
```

Вызов утилиты view аналогичен вызову редактора vim с ключом `-R` (readonly — только для чтения). При вызове редактора в этом режиме вы не можете записывать содержимое из рабочего буфера обратно в файл, чье имя было указано в командной строке. Но вы всегда сможете записать содержимое рабочего буфера в файл с другим именем. Если у вас установлена программа mc (Midnight Commander), команда view приводит к вызову не редактора vim, а программы mcview.

Длина строки и размер файла

Редактор vim работает с файлами любого формата, обеспечивающего определенную длину отдельной строки (то есть тех символов, которые находятся между двумя разделителями строк), которая может поместиться в доступную память. Общая длина файла ограничивается только доступным пространством на диске и в памяти.

Окна

Редактор vim допускает открытие, закрытие и сокрытие нескольких окон, в каждом из которых можно редактировать разные файлы. Большинство команд для работы с окнами состоят из комбинации CONTROL+W, за которой следует какая-нибудь другая буква. Например, команда `CONTROL+W s` открывает еще одно окно (разбивая экран на части), в котором редактируется тот же самый файл. Команда `CONTROL+W n` открывает второе окно для редактирования пустого файла. Команда `CONTROL+W w` перемещает курсор между окнами, а команда `CONTROL+W q` (или `:q`) приводит к выходу из окна (к его закрытию). Чтобы вывести полный перечень команд для работы с окнами, введите команду `:help windows`.

Блокировки файлов

При редактировании существующего файла vim отображает первые несколько строк файла, выводит информацию о файле в строке состояния и блокирует файл. При попытке открыть заблокированный файл с помощью редактора vim вы увидите сообщение, похожее на то, что показано на рис. 6.7. Подобные сообщения могут появляться при двух сценариях развития событий: при попытке редактирования файла, который уже кем-то редактируется (возможно, вы редактируете его в другом окне, в фоновом режиме или на другом терминале), и когда вы пытаетесь отредактировать файл, в процессе редактирования которого произошел сбой редактора vim или всей системы.

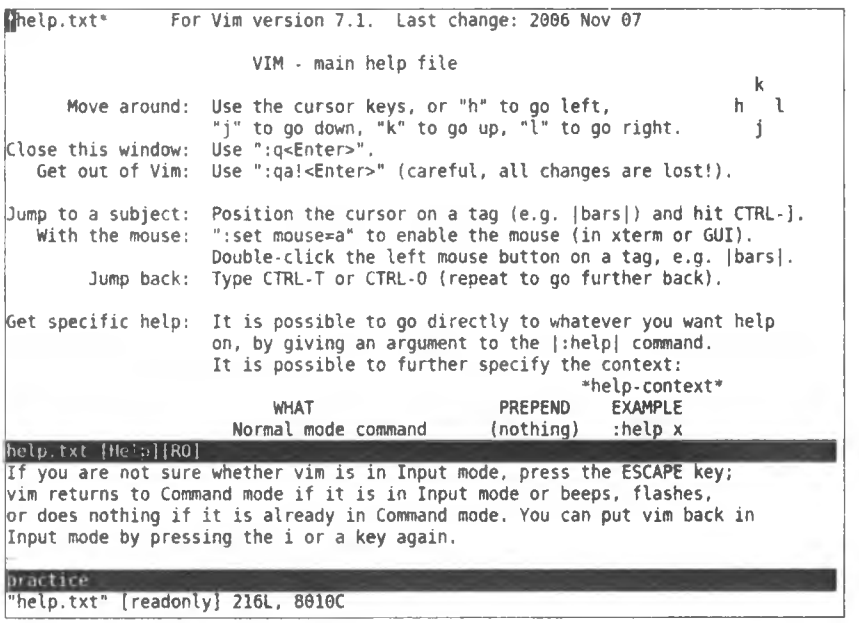

Рис. 6.7. Попытка открытия заблокированного файла

Хотя желательно следовать инструкциям, отображаемым редактором vim, второй пользователь может отредактировать файл и записать его под другим именем файла. Дополнительные сведения даны в следующем разделе.

Аварийное завершение сеанса редактирования

Сеанс редактирования можно завершить двумя способами: при выходе из vim можно сохранить изменения, внесенные в процессе сеанса редактирования, или же можно отказаться от этих изменений. Чтобы сохранить изменения и выйти из vim, можно воспользоваться командой ZZ или :wq (см. раздел «Завершение сеанса редактирования» на с. 176).

Чтобы завершить сеанс редактирования без записи содержимого рабочего буфера, нужно ввести следующую команду:

```
:q!
```

Командой :q! нужно пользоваться с особой осторожностью. Когда эта команда используется для завершения сеанса редактирования, vim не сохраняет содержимое рабочего буфера, поэтому можно потерять всю работу, проделанную со времени последней записи содержимого рабочего буфера на диск. При следующем сеансе редактирования или использования файла он появится в том виде, в котором был при последней записи рабочего буфера на диск.

Иногда может сложиться ситуация, при которой вы создали или отредактировали файл, а vim не дает возможности выйти из программы. Например, если вы забыли указать имя файла при первом вызове vim, то при вводе команды ZZ будет выдано сообщение об отсутствии имени файла («No file name»). Если vim не позволяет выйти из него обычным способом, можно воспользоваться командой Записать — Write (:w), чтобы ввести имя файла и записать его на диск перед прекращением работы vim. Введите следующую команду, подставив свое имя файла вместо элемента *имя_файла* (не забудьте после набора команды нажать клавишу ВВОД):

```
:w имя_файла
```

После ввода команды Записать (Write) можно воспользоваться командой :q, чтобы завершить работу с vim. Включать в нее восклицательный знак не нужно (как в команде q!), он нужен, только если после последней записи рабочего буфера на диск были внесены какие-нибудь дополнительные изменения. Дополнительные сведения о команде Записать (Write) даны в разделе «Запись файлов» а с. 202.

КОГДА ЗАПИСАТЬ ДАННЫЕ В ФАЙЛ НЕВОЗМОЖНО

Бывает, что нужно с помощью команды :w имя_файла записать данные в файл при отсутствии прав записи в редактируемый файл. Если ввести команду ZZ, то будет выведено сообщение о том, что файл доступен только по чтению. При этом нужно воспользоваться командой Записать (Write), указав временное имя файла, чтобы записать файл на диск под другим именем. Но если у вас нет прав записи в рабочий каталог, vim не сможет записать файл на диск. Нужно еще раз ввести команду, используя вместо имени файла абсолютное путевое имя фиктивного (несуществующего) файла в вашем личном каталоге. (Например, пользователь Max может ввести команду :w /home/max/tempor :w ~/temp.)

Если vim сообщает о существовании файла («File exists»), нужно воспользоваться командой :w! имя_файла, чтобы перезаписать существующий файл (но при этом вы должны убедиться в необходимости такой перезаписи). См. подраздел «Команда w!» на с. 203.

Восстановление текста после сбоя

Редактор vim временно сохраняет файл, с которым вы работаете, в файле подкачки. Если во время редактирования файла происходит сбой системы, текст этого файла часто удается восстановить из файла подкачки. При попытке редактирования файла,

у которого есть файл подкачки, вы увидите сообщение, похожее на то, которое показано на рис. 6.7. Если файл редактирует кто-нибудь другой, завершите работу или откройте файл в режиме «только для чтения».

В следующем примере пользователь Max использует ключ `-r`, чтобы проверить факт существования файла подкачки для файла по имени memo, который им редактировался на момент сбоя системы:

```
$ vim -r
Swap files found:             Файлы подкачки найдены:
   In current directory:          В текущем каталоге:
1.    .party.swp
          owned by: max   dated: Sat Jan 26 11:36:44 2008
         file name: ~max/party
          modified: YES
         user name: max   host name: coffee
        process ID: 18439
2.    .memo.swp
          owned by: max   dated: Mon Mar 23 17:14:05 2009
         file name: ~max/memo
          modified: no
         user name: max   host name: coffee
        process ID: 27733 (still running)
   In directory ~/tmp:           В каталоге ~/tmp:
      -- none --
   In directory /var/tmp:        В каталоге /var/tmp:
      -- none --
   In directory /tmp:            В каталоге /tmp:
-- none --
```

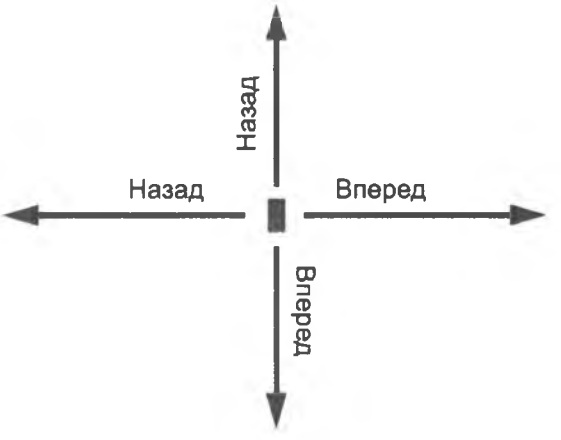

Рис. 6.8. Направления вперед и назад

При запуске с ключом `-r` редактор vim выводит список сохраненных им файлов подкачки (некоторые могут быть уже устаревшими). Если ваша работа была сохранена, введите ту же команду, затем пробел и имя файла. После этого вы вернетесь к редактированию самой последней копии рабочего буфера. Сразу же введите команду `:w` *имя_файла*, чтобы сохранить спасенную копию рабочего буфера на диске под именем, отличающимся от имени исходного файла. Затем проверьте восстановленный файл, чтобы убедиться, что все в порядке. В следующем примере показано, как пользователь

Max производит обмен данных с помощью vim при восстановлении файла memo. После этого он удаляет файл подкачки:

```
$ vim -r memo
Using swap file ".memo.swp"     Использование файла подкачки ".memo.swp"
Original file "~/memo"          Исходный файл "~/memo"
Recovery completed. You should check if everything is OK.
Восстановление завершено. Вам нужно проверить, все ли в порядке.
(You might want to write out this file under another name
(Может потребоваться запись этого файла под другим именем
and run diff with the original file to check for changes)
и запуск утилиты diff с указанием исходного файла для проверки изменений)
Delete the .swp file afterwards.
После этого файл с расширением .swp нужно удалить
Hit ENTER or type command to continue
Нажмите ВВОД или введите команду, чтобы продолжить работу
:w memo2
:q
$ rm .memo.swp
```

ВОССТАНАВЛИВАЙТЕ ФАЙЛЫ НА ТОЙ СИСТЕМЕ, КОТОРАЯ ИСПОЛЬЗОВАЛАСЬ ДО СБОЯ

Имеющаяся в vim функция восстановления применяется к той системе, которая использовалась на момент сбоя. Если вы работаете на кластере, то для успешного использования ключа –r нужно войти в ту систему, которая использовалась до возникновения сбоя.

Командный режим: Перемещение курсора

Когда vim находится в командном режиме, можно установить курсор на любой символ, имеющийся на экране. Можно также отобразить на экране другой фрагмент рабочего буфера. За счет манипуляций с экраном и с позицией курсора можно поместить курсор на любой символ, имеющийся в рабочем буфере.

Курсор можно перемещать по тексту вперед и назад. Как показано на рис. 6.8, слово «вперед» означает сдвиг в правую и в нижнюю часть экрана, а также «к концу файла», а слово «назад» означает сдвиг в левую и в верхнюю часть экрана, а также «к началу файла». При использовании команды, перемещающей курсор вперед за пределы строки (вправо), он перемещается к началу следующей строки (влево). Аналогично, когда вы перемещаете курсор назад за начало строки, он, как правило, перемещается в конец предыдущей строки.

Длинные строки

Иногда строка, находящаяся в рабочем буфере, может быть слишком длинной, чтобы уместиться в одной строке на экране. В таком случае vim переносит текущую строку на следующую строку экрана (если только не установлен параметр запрещения переноса — nowrap, см. табл. 6.7).

Можно перемещать курсор по тексту, осуществляя переход, соответствующий любой *единице отсчета* (символу, слову, строке, предложению, абзацу или экрану). Если перед командой перемещения курсора поставить число, которое называется *количеством повторений*, курсор переместится на указанное количество блоков текста. Точные определения всех этих терминов даны в разделе «Единицы отсчета» на с. 211.

Посимвольное перемещение курсора

Команды l и h

Клавиша ПРОБЕЛ вызывает посимвольное перемещение курсора вперед, по направлению к правой стороне экрана. Клавиша l (буква «l» в нижнем регистре) и клавиша СТРЕЛКА ВПРАВО (рис. 6.9) делают то же самое. Например, команда 7 ПРОБЕЛ или 7l перемещает курсор на семь символов вправо. Эти клавиши не в состоянии переместить курсор за пределы конца текущей строки на начало следующей строки. Клавиши h и СТРЕЛКА ВЛЕВО работают также, как и клавиши l и СТРЕЛКА ВПРАВО, но только перемещают курсор в обратном направлении.

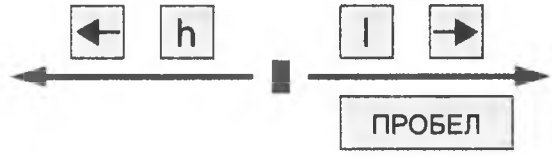

Рис. 6.9. Перемещение курсора по символам

Перемещение курсора на указанный символ

Команды f и F

С помощью команды Найти (Find) можно переместит курсор к следующему, указанному в этой команде символу, встреченному в текущей строке. Например, следующая команда перемещает курсор с его текущей позиции к следующему встреченному символу a, если таковой найдется в той же строке:

```
fa
```

Можно также найти предыдущий символ, используя заглавную букву F. Следующая команда перемещает курсор на позицию ближайшей предыдущей буквы a в текущей строке:

```
Fa
```

Точка с запятой (;) приводит к повторению последней команды Найти (Find).

Перемещение курсора по словам

Команды w и W

Клавиша w (word — слово) перемещает курсор вперед, к первой букве следующего слова (рис. 6.10). Группы символов пунктуации также считаются словами. Эта команда приводит к переходу на следующую строку, если на ней располагается следующее слово. Команда 15w приводит к перемещению курсора на первый символ пятнадцатого следующего слова.

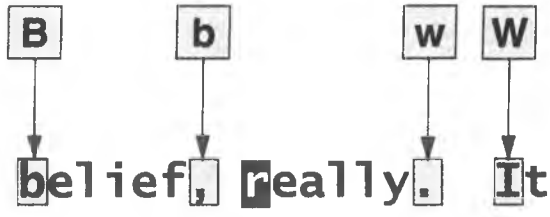

Рис. 6.10. Перемещение курсора по словам

Клавиша W похожа по действию на клавишу w, но перемещает курсор по словам, разделенным пробелами, включая и пропускаемые знаки пунктуации (см. раздел «Слово, отделенное пробелами» на с. 212.)

Команды b и B
Команды e и E

Клавиша b (back — назад) перемещает курсор назад на первую букву предыдущего слова. Клавиша B перемещает курсор назад по словам, разделенным пробелами. Аналогично этому, клавиша e перемещает курсор к концу следующего слова, а клавиша E перемещает его к концу следующего слова, отделенного пробелом.

Построчное перемещение курсора

Команды j и k

Клавиша ВВОД перемещает курсор к началу следующей строки, а клавиши j и СТРЕЛКА ВНИЗ перемещают курсор вниз на одну строку, на тот символ, который расположен непосредственно под текущим символом (рис. 6.11). Если прямо под текущим символом нет никакого символа, курсор перемещается в конец следующей строки. Курсор не будет перемещаться за пределы последней строки рабочего буфера.

Клавиши k и СТРЕЛКА ВВЕРХ работают аналогично клавишам j и СТРЕЛКА ВНИЗ, но в противоположном направлении. Клавиша минус (–) работает аналогично клавише ВВОД, но в противоположном направлении.

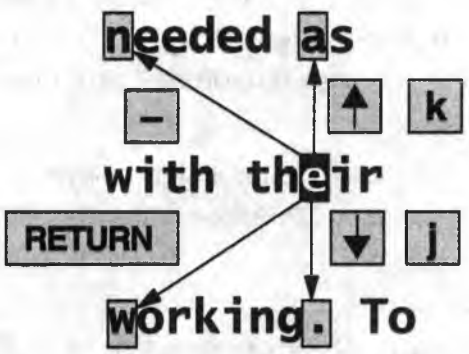

Рис. 6.11. Перемещение курсора по строкам

Перемещение курсора по предложениям и абзацам

Команды) и (
Команды } и {

Клавиши) и } перемещают курсор вперед к началу следующего предложения или следующего абзаца соответственно (рис. 6.12). Клавиши (и { перемещают курсор назад к началу текущего предложения или абзаца соответственно. Дополнительные сведения о предложениях и абзацах даны в разделах «Предложение» и «Абзац» на с. 213.

Перемещение курсора по экрану

Команды H, M и L

Клавиша H (home — в начало) перемещает курсор в крайнюю левую позицию самой верхней строки экрана, клавиша M (middle — центр) перемещает курсор на среднюю строку экрана, а клавиша L (lower — нижняя) перемещает курсор на самую нижнюю строку экрана (рис. 6.12).

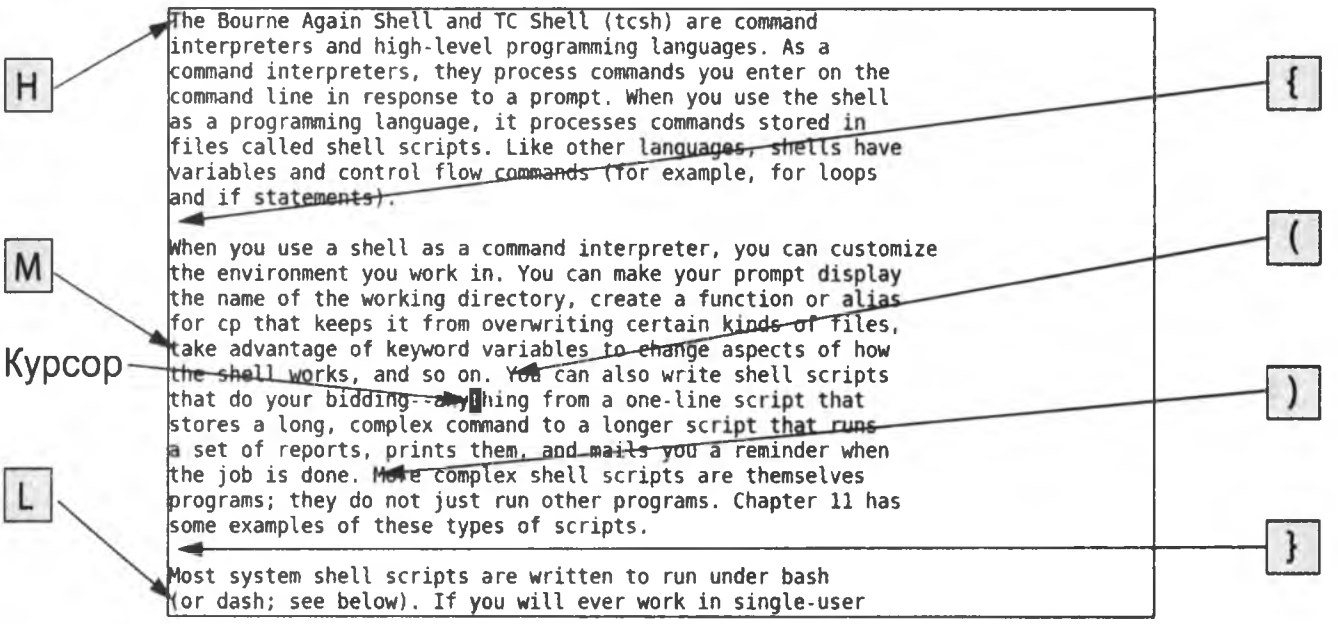

Рис. 6.12. Перемещение курсора по предложениям, абзацам и по командам H, M и L

Просмотр различных частей рабочего буфера

На экране отображается часть текста, находящегося в рабочем буфере. Используя прокрутку, можно отобразить текст, предшествующий тексту, отображенному на экране, или текст, идущий после него. Можно также отобразить фрагмент рабочего буфера, задав номер строки.

Команды CONTROL+D и CONTROL+U

Чтобы прокрутить экран вниз (вперед) по файлу, заставив vim отобразить пол-экрана нового текста, следует нажать комбинацию CONTROL+D. А чтобы прокрутить экран вверх (назад) на такое же количество строк, нужно нажать комбинацию CONTROL+U. Если перед каждой из этих команд поставить число, то vim будет прокручивать экран на указанное этим числом количество строк при каждом нажатии CONTROL+D или CONTROL+U до конца текущего сеанса (если только вы опять не измените количество прокручиваемых строк). Параметр scroll (прокрутка) рассматривается в табл. 6.7.

Команды CONTROL+F и CONTROL+B

Комбинации CONTROL+F (forward — вперед) и CONTROL+B (backward — назад) отображают на экране почти полностью новый текст, оставляя всего пару строк из содержимого прежнего экрана, чтобы не терялось представление о содержимом текста. На многих клавиатурах вместо CONTROL+F и CONTROL+B можно использовать соответственно клавиши PAGE DOWN и PAGE UP.

Номера строк (G)

Когда после номера строки вводится G (goto — перейти к), vim помещает курсор на указанную строку в рабочем буфере. Если нажать клавишу G без номера, vim установит курсор на последнюю строку рабочего буфера. В явном виде номеров строк не существует, поэтому, чтобы команда работала, файлу не нужно иметь нумерацию строк. Если нужно, чтобы vim показывал номера строк, обратитесь к параметру «Отображение номеров строк» в табл. 6.7 на с. 206.

Режим ввода

Команды Вставить (Insert), Добавить (Append), Открыть (Open), Изменить (Change) и Заменить (Replace) переводят vim в режим ввода. Когда vim находится в этом режиме, вы можете вводить новый текст в рабочий буфер. Чтобы по завершению набора текста вернуть vim в командный режим, следует нажать клавишу ESCAPE. Если нужно, чтобы vim напоминал о том, что вы находитесь в режиме ввода, обратитесь к параметру «Отображение режима» в табл. 6.7 (он делает это по умолчанию) на с. 207.

Вставка текста

Команды Вставить (i и I)

Команда i (Insert — Вставить) переводит vim в режим ввода и помещает введенный текст перед текущим символом. Команда I помещает текст в начале текущей строки (рис. 6.13). Хотя иногда команды i и I перезаписывают текст на экране, символы, находящиеся в рабочем буфере, при этом не изменяются, этот эффект оказывает влияние только на отображение текста. Перезаписанный текст снова появляется на экране при нажатии клавиши ESCAPE и возвращении редактора vim в командный режим. Команда i или I используется для вставки нескольких символов или слов в существующий текст или для вставки текста в новый файл.

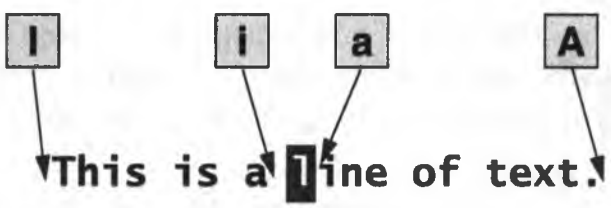

Рис. 6.13. Команды I, i, a и A

Добавление текста

Команды Добавить (a и A)

Команда a (Append — Добавить) похожа на команду i, за исключением того, что она помещает текст после текущего символа (см. рис. 6.13). Команда A помещает текст после последнего символа текущей строки.

Открытие строки для текста

Команды Открыть (o и O)

Команда o (Open — Открыть) открывает пустую строку внутри существующего текста, помещает курсор на начало новой (пустой) строки и переводит vim в режим ввода. Команда O открывает строку над текущей строкой, а команда o открывает строку под ней. Эти команды используются при вводе нескольких новых строк вовнутрь уже существующего текста.

Замена текста

Команды Заменить (r и R)

Команды r и R (Replace — Заменить) заставляют переписать (заменить) существующий текст введенным вами текстом. Один символ, введенный перед вводом команды r, переписывает текущий символ. После ввода этого символа vim возвращается в командный режим, не требуя для этого нажатия клавиши ESCAPE.

Команда R заставляет переписывать текст всеми последующими символами, пока вы не нажмете клавишу ESCAPE, чтобы вернуть vim в командный режим.

ЗАМЕНА ТАБУЛЯЦИЙ

Бывает, что команды замены ведут себя при замене символов табуляции несколько странно. Еще до попытки их замены символы табуляции могут появляться в виде нескольких пробелов. Сам по себе символ табуляции существует в виде одного символа и заменяется одним символом. Сведения о том, как превратить символы табуляции в символы, отображаемые на экране, даны в табл. 6.7, в описании параметра «Отображение невидимых символов» на с. 206.

Цитирование специальных символов в режиме ввода

Команда CONTROL+V

Работая в режиме ввода, можно применять команду Цитировать (Quote), CONTROL+V, для ввода любого символа в текст, включая символы, которые обычно имеют для vim особое значение. Среди них символ, генерируемый комбинацией CONTROL+L (или CONTROL+R), который вызывает перерисовку экрана; символ, генерируемый комбинацией CONTROL+W, возвращающий курсор назад на слово влево; разделитель строк, генерируемый комбинацией CONTROL+M, и символ, генерируемый клавишей ESCAPE, который завершает работу в режиме ввода.

Чтобы вставить один из этих символов в текст, нужно сначала ввести символ, генерируемый комбинацией CONTROL+V, а затем специальный символ. Комбинация CONTROL+V приводит к цитированию одного следующего за ней символа. Например, для ввода последовательности ESCAPE[2J в файл, создаваемый в редакторе vim, нужно набрать символьную последовательность с помощью клавиш CONTROL+V ESCAPE[2J. Эта последовательность приводит к очистке экрана терминала DEC VT-100 и других, похожих на него терминалов. Хотя обычно вводить эту последовательность в документ и не требуется, ее использование (или использование других ESCAPE-последовательностей) может пригодиться в сценариях оболочки, создаваемых в редакторе vim. Создание сценариев оболочки рассматривается в главе 10.

Командный режим: Удаление и изменение текста

В этом разделе рассматриваются команды, предназначенные для удаления и замены или изменения текста в редактируемом документе. Команда Undo (Отменить) рассматривается здесь по той причине, что она позволяет восстановить удаленный или измененный текст.

Отмена изменений

Команды Отменить (u и U)

Команда u (Undo — Отменить) восстанавливает только что ошибочно удаленный или измененный текст. Одиночная команда Отменить восстанавливает только текст, удаленный самым последним действием. Если вы удалили строку, а затем изменили слово, то первая команда Отменить восстанавливает только измененное слово, а чтобы восстановить удаленную строку, нужно ввести еще одну команду Отменить. При установленном параметре совместимости — compatible (см. раздел «Параметр compatible» на с. 177), vim может отменить только самые последние изменения. Команда U восстанавливает последнюю измененную строку, возвращая ее к состоянию, предшествующему изменениям, даже если в нее вносилось сразу несколько изменений.

Удаление символов

Команды Удалить символ (x и X)

Команда x удаляет текущий символ. Перед вводом команды можно поставить количество повторений (см. раздел «Количество повторений» на с. 214), чтобы удалить сразу несколько символов в текущей строке, начиная с текущего символа. Команда X удаляет символ слева от курсора.

Удаление текста

Команды Удалить (d и D)

Команда d (Delete — Удалить) удаляет текст из рабочего буфера. Объем текста, удаляемого командой d, зависит от количества повторений и от единиц отсчета (см. раздел «Единицы отсчета» на с. 211). После удаления текста vim продолжает находиться в командном режиме.

ДЛЯ УДАЛЕНИЯ СТРОКИ ПОЛЬЗУЙТЕСЬ КОМАНДОЙ DD

Команда d ВВОД удаляет две строки: текущую и следующую за ней строку. Чтобы удалить только текущую строку, используйте команду dd или поставьте перед командой dd количество повторений (см. раздел «Количество повторений» на с. 214), чтобы удалить сразу несколько строк.

Можно удалить текст от текущей позиции курсора и до указанного символа в той же строке. Чтобы удалить текст до следующей точки с запятой (;), введите команду dt; (дополнительные сведения о команде t даны в разделе «Команды Найти (t и T)» на с. 193). Чтобы удалить текст до конца текущей строки, воспользуйтесь командой D или d$. В табл. 6.1 приводится список некоторых команд Удалить. Каждая команда, за исключением последней группы, начинающейся с команды dd, производит удаление от или до текущего символа.

ПЕРЕСТАНОВКА МЕСТАМИ СИМВОЛОВ И СТРОК

Если два символа стоят не в том порядке, установите курсор на первый из них и введите команду xp. Если нарушен порядок следования двух строк, установите курсор на первую из них и введите команду ddp.

Дополнительные сведения о командах Поместить (Put) даны в разделе «Команды Поместить (p и P)» на с. 200.

Таблица 6.1. Примеры команд Удалить

Команда	Результат
d1	Удаляет текущий символ (аналог команды x)
d0	Удаляет от начала строки
d^	Удаляет от первого символа строки (не включая лидирующие пробелы или символы табуляции)
Dw	Удаляет до конца слова
d3w	Удаляет до конца третьего слова
Db	Удаляет от начала слова
dW	Удаляет до конца слова, отделенного пробелом
dB	Удаляет от начала слова, отделенного пробелом
d7B	Удаляет от начала седьмого предыдущего слова, отделенного пробелом

Команда	Результат
d)	Удаляет до конца предложения
d4)	Удаляет до конца четвертого предложения
d(	Удаляет от начала предложения
d}	Удаляет до конца абзаца
d{	Удаляет от начала абзаца
d7{	Удаляет от начала седьмого предыдущего абзаца
d/text	Удаляет до следующего встреченного слова text
Dfc	Удаляет текст в текущей строке вплоть до следующего встреченного символа c, включая сам символ
Dtc	Удаляет текст в текущей строке вплоть до следующего встреченного символа c
D	Удаляет до конца строки
d$	Удаляет до конца строки
Dd	Удаляет текущую строку
5dd	Удаляет пять строк, начиная с текущей
dL	Удаляет до последней строки экрана
dH	Удаляет с первой строки экрана
dG	Удаляет до конца рабочего буфера
d1G	Удаляет от начала рабочего буфера

Изменение текста

Команды Изменить (c и C)

Команда c (Change — Изменить) заменяет существующий текст новым текстом, который не обязательно должен занимать то же место, что и существующий. Можно заменить одно слово несколькими, одну строку несколькими строками или заменить целый абзац одним символом. Команда C заменяет текст от текущей позиции курсора до конца строки.

Команда c удаляет то количество текста, которое определено количеством повторений и единицей отсчета (см. раздел «Единицы отсчета» на с. 211), и переводит vim в режим ввода. Когда после ввода нового текста будет нажата клавиша ESCAPE, старое слово, строка, предложение или абзац будут заменены новым содержимым. Нажатие клавиши ESCAPE без ввода нового текста приводит к удалению указанного текста (то есть команда заменяет указанный текст пустой строкой).

В табл. 6.2 приводится список некоторых команд Изменить. За исключением последних двух команд, каждая команда заменяет текст от или до текущего символа.

Таблица 6.2. Примеры команд Изменить

Команда	Результат
cl	Изменяет текущий символ
cw	Изменяет символы до конца слова
c3w	Изменяет символы до конца третьего слова
cb	Изменяет символы с начала слова
cW	Изменяет символы до конца слова, отделенного пробелом

продолжение ↛

Таблица 6.2 (*продолжение*)

Команда	Результат
cB	Изменяет символы от начала слова, отделенного пробелом
C7B	Изменяет символы с начала предыдущего седьмого слова, отделенного пробелом
C$	Изменяет символы до конца строки
C0	Изменяет символы от начала строки
c)	Изменяет символы до конца предложения
C4)	Изменяет символы до конца четвертого предложения
C(	Изменяет символы от начала предложения
C}	Изменяет символы до конца абзаца
C{	Изменяет символы от начала абзаца
C7{	Изменяет символы от начала предыдущего седьмого абзаца
ctc	Изменяет символы в текущей строке до следующего встреченного символа C
C	Изменяет символы до конца строки
cc	Изменяет текст текущей строки
5cc	Изменяет текст пяти строк, начиная с текущей строки

КОМАНДА DW ОТЛИЧАЕТСЯ ОТ КОМАНДЫ CW

Команда dw удаляет все символы до конца слова до пробела (включительно). Команда cw изменяет только символы в слове, не касаясь замыкающего пробела.

Замена текста

Команды Подставить (s и S)

Команды s и S (Substitute — Подставить) также заменяют существующий текст новым текстом (табл 6.3). Команда s удаляет текущий символ и переводит vim в режим ввода. Она подставляет вместо текущего символа тот текст, который вводится до нажатия клавиши ESCAPE. Команда S делает то же самое, что и команда cc: она заменяет текущую строку. Команда s подставляет символы только в текущей строке. Если перед командой s указать количество повторений, причем это действие должно будет подставить больше символов, чем имеется в текущей строке, то команда s подставит символы только до конца строки (так же, как и команда C).

Таблица 6.3. Примеры команд Подставить

Команда	Результат
s	Подставляет вместо текущего символа один или несколько символов
S	Подставляет вместо текущей строки один или несколько символов
5s	Подставляет вместо пяти символов, начиная с текущего, один или несколько символов

Изменение регистра

Знак тильда (~) изменяет регистр текущего символа с верхнего на нижний, и наоборот. Можно указать перед знаком тильды количество символов, на которое нужно

распространить действие команды. Например, команда 5~ преобразует следующие пять символов, начиная с текущего (который находится под курсором), но не меняет символы, которые следуют после конца текущей строки.

Поиск и замена

Поиск и замена символа, строки текста или строки, соответствующей регулярному выражению, является ключевым свойством любого редактора. Редактор vim использует простые команды для поиска символа в текущей строке. Он также содержит более сложные команды для поиска и, в качестве дополнительной функции, замены одного или нескольких встреченных строчных значений или фрагментов текста, соответствующих регулярным выражениям, другим строчным значениям по всему объему рабочего буфера.

Поиск символа

Команды Найти (f и F)

Для поиска указанного символа и перемещения курсора на следующий такой же встреченный символ в текущей редактируемой строке используется команда f (Find — Найти). Дополнительные сведения о ней даны в разделе «Перемещение курсора на указанный символ» на с. 185.

Команды Найти (t и T)

Следующие две команды используются аналогично командам Найти. Команда t помещает курсор на символ, стоящий перед следующим указанным символом. Команда T помещает курсор на символ, стоящий после предыдущего указанного символа.

Ввод точки с запятой (;) приводит к повторному выполнению последней команды f, F, t или T.

Эти команды поиска могут применяться в сочетании с другими командами. Например, команда d2fq приводит к удалению текста от текущего символа до второй встреченной буквы q в текущей строке.

Поиск строки

Команды Искать (/ и ?)

Редактор vim может искать в рабочем буфере строку текста или строку, соответствующую регулярному выражению как в прямом, так и в обратном направлении. Для поиска следующего присутствия строки в буфере (в прямом направлении) следует нажать клавишу прямого слэша (/), ввести текст, который нужно найти (называемый *строкой поиска*), и нажать клавишу ВВОД. После нажатия клавиши слэша vim отображает символ слэша в строке состояния. По мере ввода строки текста, она также отображается в строке состояния. После нажатия клавиши ВВОД vim ищет строку в тексте. Если этот поиск завершается успешно, vim устанавливает курсор на первый символ найденной строки. Если вместо прямого слэша используется вопросительный знак (?), vim ищет предыдущее присутствие строки в буфере. Чтобы включить прямой слэш в строку поиска в прямом направлении или вопросительный знак в строку поиска в обратном направлении, перед ними следует поставить символ обратного слэша (\).

ДВА РАЗНЫХ СПОСОБА ЦИТИРОВАНИЯ СИМВОЛОВ

Для цитирования специальных символов в тексте, вводимом в файл, используется сочетание клавиш CONTROL+V (см. раздел «Цитирование специальных символов в режиме ввода» на с. 189). В этом разделе для простого цитирования специальных символов в строке поиска используется обратный слэш (\ ). Эти две технологии цитирования символов не являются взаимозаменяемыми.

Команды Следующий (n и N)

Клавиши N и n используются для повторения последнего поиска и не требуют повторного ввода строки поиска. При использовании клавиши n происходит точное повторение исходного поиска, а при использовании клавиши N поиск повторяется в направлении, противоположном исходному.

Если поиск производится в прямом направлении и редактор vim не находит строку поиска, достигнув конца рабочего буфера, редактор обычно *зацикливает поиск* и продолжает его с начала буфера. При проведении поиска в обратном направлении vim также зацикливает поиск и продолжает его с начала рабочего буфера и до его конца. Кроме этого, при поиске vim, как правило, чувствителен к регистру букв. Сведения о том, как изменить эти параметры, даны в табл. 6.7, в описании «Продолжение сканирования» на с. 207 и «Игнорирование регистра при проведении поисков» на с. 206.

Сравнение обычного поиска и поиска с последовательным уточнением

Чтобы vim провел обычный поиск (проводимый по умолчанию), нужно ввести слэш (/) или вопросительный знак (?), затем ввести строку поиска, после чего нажать клавишу ВВОД. После этого редактор vim перемещает курсор на следующую или предыдущую встреченную им искомую строку.

Когда vim ведет поиск с последовательным уточнением, вы вводите / или ?. По мере ввода каждого символа строки поиска, vim перемещает выделение к следующему или к предыдущему встреченному строковому значению, соответствующему текущему содержимому строки поиска. Когда будет выделена искомая строка, нужно нажать клавишу ВВОД, чтобы курсор переместился на это выделение. Если введенная строка не соответствует ни одному из фрагментов текста, редактор vim ничего не выделяет.

Тип проводимого редактором vim поиска зависит от параметра `incsearch`. Чтобы включить поиск с последовательным уточнением, нужно ввести команду `:set incsearch`, а для его выключения используется команда `noincsearch`. При установке параметра `compatible` (см. раздел «Параметр compatible» на с. 177) редактор vim выключает поиск с последовательным уточнением.

Использование специальных символов в строках поиска

Поскольку строка поиска является регулярным выражением, некоторые, вводимые в нее символы приобретают специальное значение. В следующих абзацах перечисляется ряд таких символов.

Первые два элемента следующего списка (^ и $) всегда имеют специальные значения в строке поиска, если только они не будут процитированы путем установки перед ними обратного слэша (\). Для всех остальных элементов списка их специальное значение можно отключить путем установки параметра `nomagic`. Дополнительная информация дана в табл. 6.7, в описании параметра «Разрешение применения специальных символов в строках поиска» на с. 205.

^ Индикатор начала строки

Когда первым символом в строке поиска является знак вставки (называемый также диакритическим знаком), он соответствует началу строки. Например, команда `/^the` задает поиск следующей редактируемой строки, начинающейся с символьной строки `the`.

$ Индикатор конца строки

Знак доллара означает окончание строки. Например, команда `/!$` задает поиск следующей строки, которая оканчивается восклицательным знаком, а команда `/ $` задает поиск следующей строки, оканчивающейся пробелом.

. Индикатор любого символа

Точка соответствует *любому* символу, находящемуся в любом месте строки поиска. Например, команда `/l..e` находит символьные строки `line`, `followed`, `like`, `included`, `all memory` или любое другое слово или символьную строку, в котором содержится буква l, за которой следуют любые два символа и буква e. Для поиска самой точки ее нужно процитировать с помощью обратного слэша (`\.`).

\> Индикатор конца слова

Эта пара символов соответствует концу слова. Например, команда `/s\>` находит следующее слово, оканчивающееся на букву s. Несмотря на то что обратный слэш (\), как правило, используется для отключения специального значения символа, последовательность символов \> имеет специальное значение, а сам по себе символ > такого значения не имеет.

\< Индикатор начала слова

Эта пара символов соответствует началу слова. Например, команда `/\<The` находит следующее слово, которое начинается со строки `The`. Индикатор начала слова использует обратный слэш таким же необычным образом, как и индикатор конца слова.

* Признак нулевого и любого другого количества появлений

Этот символ служит модификатором, который будет соответствовать нулевому и любому другому количеству появлений того символа, за которым он непосредственно следует. Например, команда `/dis*m` будет соответствовать строке `di`, за которой следуют от нуля и более символов `s`, за которыми следует символ `m`. Примерами успешно найденного соответствия может послужить строка `dim` или `dism`, а также строка `dissm`.

[] Определение класса символов

Квадратные скобки, в которые заключены от двух и более символов, соответствуют любому *одному* символу из числа тех, что заключены в эти скобки. Например, команда `/dis[ck]` находит следующее появление слова `disk` *либо* слова `disc`.

Внутри определения класса можно использовать два специальных символа. Знак вставки (^), который используется в качестве первого символа, следующего сразу за левой квадратной скобкой, определяющей класс символов, соответствует *любому символу, за исключением тех, что следуют за ним*. Знак дефиса, поставленный между двумя символами, служит индикатором диапазона символов. Примеры приведены в табл. 6.4.

Таблица 6.4. Примеры поиска

Строка поиска	Что она находит
`/and`	Находит следующее появление строки **and** Примеры: **sand and standard slander andiron**

продолжение ⇩

Таблица 6.4 (*продолжение*)

Строка поиска	Что она находит
/ \<and\>	Находит следующее появление слова **and** Пример: **and**
/^ The	Находит следующую строку, начинающуюся со значения **The** Примеры: **The...** **There...**
/^[0-9][0-9])	Находит следующую строку, начинающуюся с двузначного номера, за которым следует правая круглая скобка Примеры: **77)...** **01)...** **15)...**
/ \<[adr]	Находит следующее слово, начинающееся с **a**, **d** или **r** Примеры: **apple drive road argument right**
/^[A-Za-z]	Находит следующую строку, начинающуюся с буквы в верхнем или нижнем регистре Примеры: **эта команда не найдет строку, начинающуюся с цифры 7...** **Дорогой мистер Джонс...** **находит строку, являющуюся, как и эта, частью предложения...**

Подстановка одной строки вместо другой

Команда Подставить сочетает в себе действия команд Искать и Изменить. То есть она ведет поиск строки, соответствующей регулярному выражению, точно так же, как команда /, допуская использование тех же самых специальных символов, которые были рассмотрены в предыдущем разделе. Когда найдена строка, соответствующая регулярному выражению, команда Изменить вносит в нее изменения. Синтаксис команды Подставить имеет следующий вид

:[*g*][*адрес*]s/*строка_поиска*/*строка_замены*[/*ключ*]

Поскольку команда Подставить начинается с двоеточия, vim выполняет ее, как и все такие команды, из строки состояния.

Адрес подстановки

Если не указать *адрес*, команда Подставить просматривает только текущую строку. Если в качестве *адреса* использовать номер отдельной строки, то команда Подставить будет вести поиск только в этой строке. Если в качестве *адреса* указать номера двух строк, разделенные запятой, команда Подставить будет вести поиск в этих двух строках и во всех строках, расположенных между ними. Если вы хотите, чтобы vim отображал номер строк, установите параметр «Отображение номеров строк», рассмотренный в табл. 6.7 на с. 206. Везде, где в качества *адреса* можно использовать номер строки, можно также использовать и *адресную* строку, заключенную в слэши. В таком случае редактор vim работает со следующей найденной строкой, соответствующей *адресной*

строке. Когда перед первым слэшем адресной строки стоит буква *g* (указывающая на глобальный характер замены), vim обрабатывает все строки в файле, соответствующие *адресной* строке. (Эта буква *g* отличается от той, которая ставится в конце команды Подставить, заставляя эту команду проводить множественные замены в одной строке; см. далее раздел «Поиск и замена строк» на с. 197).

Когда внутри *адреса* используется точка, она обозначает текущую строку, знак доллара обозначает последнюю строку рабочего буфера, а знак процента обозначает весь рабочий буфер. Можно также воспользоваться *адресной* арифметикой, используя знаки плюса и минуса. В табл. 6.5 показаны несколько примеров *адресов*.

Таблица 6.5. Адреса

Адрес	Адресуемая часть рабочего буфера
5	Строка 5
77,100	Строки с 77 по 100 включительно
1,.	С начала рабочего буфера и до текущей строки
.,$	От текущей строки и до конца рабочего буфера
1,$	Весь рабочий буфер
%	Весь рабочий буфер
/pine/	Следующая строка, содержащая слово **pine**
g/pine/	Все строки, содержащие слово **pine**
.,.+10	От текущей строки и следующие десять строк (всего 11)

Поиск и замена строк

Буква s, используемая в синтаксисе команды после *адреса*, показывает, что это команда Подставить. Разделитель, следующий за s, обозначает начало *строки_поиска*.

Хотя в примерах, приводимых в данной книге, используется прямой слэш, в качестве разделителя можно использовать любой другой символ, кроме буквы, цифры, пустого символа или обратного слэша. В конце *строки_поиска* нужно поставить тот же самый разделитель.

Затем следует *строка_поиска*. У нее такой же формат, как и у строки поиска команды /, и она может включать те же специальные символы (см. раздел «Использование специальных символов в строках поиска»на с. 194). (*Строка_поиска* является регулярным выражением.) Следующий разделитель обозначает конец *строки_поиска* и начало *строки_замены*.

Строка_замены предназначена для установки вместо текста, соответствующего *строке_поиска*, и за ней обычно следует разделительный символ. Если за *строкой_замены* не следует какой-нибудь *ключ*, то завершающий разделительный символ можно опустить, он нужен только в случае использования ключа.

Некоторые символы имеют специальные значения в *строке_поиска*, а другие символы имеют специальные значения в *строке_замены*. Например, знак амперсанда (&) в *строке_замены* представляет собой текст, который соответствовал *строке_поиска*. Обратный слэш в *строке_замены* является признаком цитирования следующего за ним символа. Дополнительные сведения содержатся в табл. 6.6.

Таблица 6.6. Примеры поиска и замены

Команда	Результат
`:s/bigger/biggest/`	Заменяет первое встреченное строковое значение **bigger** в текущей строке строковым значением **biggest** Пример: **bigger ▸ biggest**
`:1,.s/Ch 1/Ch 2/g`	Заменяет каждое встреченное до текущей строки или в ней самой строковое значение **Ch 1** строковым значением **Ch 2** Примеры: **Ch 1 ▸ Ch 2** **Ch 12 ▸ Ch 22**
`:1,$s/ten/10/g`	Заменяет каждое встреченное строковое значение **ten** строковым значением **10** Примеры: **ten ▸ 10** **often ▸ of10** **tenant ▸ 10ant**
`:g/chapter/s/ten/10/`	Заменяет первое встреченное строковое значение **ten** строковым значением **10** во всех строках, содержащих слово **chapter** Примеры: **chapter ten ▸ chapter 10** **chapters will often ▸ chapters will of10**
`:%s/ \<ten\>/10/g`	Заменяет каждое встреченное слово **ten** строковым значением **10** Пример: **ten ▸ 10**
`:.,.+10s/every/each/g`	Заменяет каждое встреченное строковое значение **every** строковым значением **each** в текущей и в следующих десяти строках Примеры: **every ▸ each** **everything ▸ eachthing**
`:s/ \<short\>/ "&"/`	Заменяет слово **short** в текущей строке строковым значением **«short»** (заключенным в кавычки) Пример: **the shortest of the short ▸ the shortest of the «short»**

Обычно команда Подставить ставит *строку_замены* только вместо первого встреченного фрагмента текста, соответствующего *строке_поиска*. Если нужна глобальная подстановка — то есть если вы хотите заменить все встреченные в строке фрагменты текста, соответствующие *строке_поиска*, — добавите после последнего разделителя, закрывающего *строку_замены*, ключ `g` (global). Еще один полезный ключ — `c` (check — проконтролировать), заставляет vim запрашивать о желании внести изменения при каждой встрече текста, соответствующего *строке_поиска*. При нажатии клавиши `y` происходит замена найденного строкового значения, нажатие клавиши `q` прерывает выполнение команды, нажатие клавиши `l` (last — последняя) приводит к замене и завершению выполнения команды, нажатие клавиши `a` (all — все) приводит к замене всех найденных далее значений, нажатие клавиши `n` (next — дальше) приводит к продолжению поиска без проведения текущей замены.

Строка *адрес* должна отличаться от *строки_поиска*. Например, команда

```
:/candle/s/wick/flame/
```

подставляет строковое значение `flame` вместо первого встреченного строкового значения `wick` на первой же следующей строке, содержащей строковое значение `candle`.

Точно так же команда

```
:g/candle/s/wick/flame/
```

осуществляет такую же подстановку вместо первого встреченного строкового значения `wick` в каждой строке файла, содержащей строковое значение `candle`, а команда

```
:g/candle/s/wick/flame/g
```

осуществляет такую же подстановку всех встреченных строковых значений `wick` на каждой строке, содержащей строковое значение `candle`.

Если *строка_поиска* будет совпадать с *адресом*, то *строку_поиска* можно оставлять пустой. Например, команда `:/candle/s//lamp/` эквивалентна команде `:/candle/s/candle/lamp/`.

Другие команды

В этом разделе рассматриваются три команды, которые трудно отнести к какой-нибудь конкретной группе.

Команда Присоединить

Команда Join (J)

Команда J (Join — Присоединить) присоединяет строку, которая находится под текущей строкой, к концу текущей строки, вставляя пробел между тем, что раньше было двумя строками, и оставляет курсор на этом пробеле. Если текущая строка заканчивалась точкой, vim вставляет два пробела. Вы всегда сможете «разъединить» (разбить) строку на две строки, заменив пробел или пробелы там, где пожелаете разбить строку символом, который сгенерирует клавиша ВВОД.

Команда Статус

Команда Status (CONTROL+G)

Команда Status, CONTROL+G, отображает имя редактируемого файла, информацию о том, был ли этот файл изменен, или это файл только для чтения, номер текущей строки, общее количество строк в рабочем буфере и процентное отношение, приходящееся на позицию предыдущей строки по отношению ко всему объему рабочего буфера. Для отображения информации состояния можно также воспользоваться командой `:f`. Следующая строка является примером строки состояния:

```
"/usr/share/dict/words" [readonly] line 28501 of 98569 --28%-- col 1
```

Команда .

Команда .

Команда . (точка) приводит к повторению самой последней команды, изменившей текст. Если, к примеру, только что была введена команда d2w (удалить следующие два слова), то команда . удалит следующие два слова. Если вы только что вставили текст, команда . повторно вставит точно такой же текст. Эта команда пригодится при изменении некоторых появлений слова или фразы в рабочем буфере. Найдите первое появление слова (с помощью команды /), а затем внесите нужные изменения (с помощью команды cw). Затем можно будет использовать команду n для поиска следующего появления этого слова, а команду . для внесения с него такого же изменения. Если вы не хотите вносить изменение, введите опять команду n, чтобы найти следующее появление этого слова.

Копирование, перемещение и удаление текста

Редактор vim имеет универсальный буфер и 26 именованных буферов, в которых может храниться текст во время сеанса редактирования. Эти буферы используются, когда нужно переместить или скопировать фрагмент текста в другое место рабочего буфера. Удалить текст из одного места в рабочем буфере и поместить его в другое место этого буфера позволяют команды Удалить и Поместить. Команды Выбрать и Поместить копируют текст в другое место рабочего буфера, не изменяя при этом исходный текст.

Универсальный буфер

Редактор vim сохраняет текст, который только что был изменен, удален или выбран (это понятие рассматривается ниже) в универсальный буфер. Команда Отменить при восстановлении текста извлекает его из универсального буфера.

Копирование текста в буфер

Команды Выбрать (y и Y)

Команда Выбрать (y) идентична команде Удалить (d), за исключением того, что она не удаляет текст из рабочего буфера. Редактор vim помещает копию выбранного текста в универсальный буфер. После этого, чтобы поместить еще одну копию выбранного фрагмента в какое-нибудь другое место рабочего буфера, используется команда Поместить. Команда Выбрать используется точно так же, как команда Удалить. При использовании буквы Y верхнего регистра выбранной и помещенной в универсальный буфер оказывается вся строка.

ЧТОБЫ ВЫБРАТЬ ОДНУ СТРОКУ, ИСПОЛЬЗУЙТЕ КОМАНДУ YY

Точно так же, как команда d ВВОД удаляет две строки, команда y ВВОД выбирает две строки. Чтобы выбрать текущую строку, используйте команду yy, а чтобы ее удалить — команду dd.

D РАБОТАЕТ НЕ ТАК КАК Y

Команда D (см. раздел «Команды Удалить (d и D)» на с. 190 работает не так, как команда Y. Команда D удаляет все до конца строки, а команда Y выбирает всю строку, независимо от позиции курсора.

Копирование текста из буфера

Команды Поместить (p и P)

Команды Поместить — p и P, копируют текст из универсального буфера в рабочий буфер. При удалении или выделении символов или слов они попадают в универсальный буфер, команда p вставляет их после текущего символа, а команда P вставляет их перед этим символом. При удалении или выделении строк, предложений или абзацев команда P вставляет содержимое универсального буфера в текущую строку, а команда p вставляет их после текущей строки.

Команды Поместить не уничтожают содержимое универсального буфера. Поэтому один и тот же текст можно поместить в несколько мест файла, если ввести одну команду Удалить или Выделить и несколько команд Поместить.

Удаление текста приводит к его копированию в буфер

Любая из рассмотренных ранее команд Удалить (см. раздел «Командный режим: Удаление и изменение текста» на с. 189) помещает удаленный текст в универсальный буфер. Точно так же, как при использовании команды Отменить, удаленный текст

можно вернуть на его прежнее место, с помощью команды Поместить можно поместить удаленный текст в другое место рабочего буфера.

Предположим, вы удаляете слово из середины предложения путем ввода команды dw, а затем перемещаете курсор на пробел между двумя словами и вводите команду p. Редактор vim помещает только что удаленное слово в новое место. Если удалить строку с помощью команды dd, а затем переместить курсор на строку ниже той, на которой нужно, чтобы появилась удаленная строка, и ввести команду P, редактор vim поместит строку на новое место.

Именованные буферы

С командами Удалить, Выбрать или Поместить можно использовать именованный буфер. Для каждого из 26 именованных буферов в качестве имен используются буквы английского алфавита. В каждом именованном буфере могут храниться разные блоки текста, и каждый блок можно вызывать повторно по мере необходимости. В отличие от универсального буфера, vim не изменяет содержимое именованного буфера до тех пор, пока вы не введете команду перезаписи этого буфера. Редактор vim сохраняет содержимое именованных буферов на время всего сеанса редактирования. Он сохраняет текст в именованном буфере, если перед командой Удалить или Выбрать поставить двойную кавычку и имя буфера (например, команда "kyy копирует текущую строку в буфер k). Информацию из рабочего буфера можно поместить в именованный буфер двумя способами. Во-первых, если в качестве имени буфера указать букву в нижнем регистре, vim перепишет содержимое этого буфера при удалении или выборке текста в него. Во-вторых, если в качестве имени буфера использовать букву в верхнем регистре, vim добавит только что удаленный или выбранный текст к концу буфера. Это свойство позволяет создавать коллекцию блоков текста из разных мест файла и размещать эти блоки в одном месте файла с помощью одной команды. Именованные буферы можно также использовать при перемещении части файла, когда не хочется вводить команду Поместить сразу же после ввода соответствующей команды Удалить и когда нужно повторно вставить в документ абзац, предложение или фразу.

Если есть предложение, использующееся по всему документу, можно выбрать это предложение в именованный буфер и поместить его везде, где нужно, используя такую процедуру: после ввода первого экземпляра этого предложения и нажатия клавиши ESCAPE для возвращения в командный режим оставьте курсор на строке, содержащей предложение. (Чтобы эта процедура сработала, предложение должно находиться отдельно на строке или строках.) Затем нужно выбрать предложение в именованный буфер, для чего ввести команду "ayy (или "a2yy, если предложение занимает две строки). Теперь как только понадобится это предложение, можно вернуться в командный режим и ввести команду "ap, чтобы поместить копию предложения ниже текущей строки.

Эта технология предоставляет быстрый и легкий способ вставки текста, который часто встречается в документе. Например, при редактировании юридического документа можно сохранить фразу «Истец утверждает, что ответчик» в именованном буфере и избавиться от необходимости набора этой фразы во всех местах, где ее предстоит использовать. Точно так же при составлении письма, в котором часто используется длинное название компании, например National Standards Institute, можно пометить это название в именованный буфер.

Нумерованные буферы

Вдобавок к 26 именованным и одному универсальному буферу, в vim доступны еще и 9 нумерованных буферов. Эти буферы в некотором смысле являются буферами только для чтения. Редактор vim заполняет их девятью последними удаленными фрагментами текста, в которых содержится хотя бы одна строка. Самый последний удаленный текст хранится в буфере "1, следующий из недавно удаленных фрагментов текста хранится в буфере "2 и т. д. Если вы удаляете блок текста, а затем вводите другие vim-команды, то есть уже не можете вернуть удаленный текст с помощью команды Отменить, то для вставки самого последнего удаленного блока текста ниже

того места, где находится курсор, можно воспользоваться командой "1p. Если вы удалили несколько блоков текста и хотите вернуть только один, конкретный блок, нужно сделать следующее: вставить с помощью команды "1p содержимое первого буфера. Если первый буфер не содержит нужный текст, отмените операцию вставки, а затем введите команду точка (.), чтобы повторить предыдущую команду. Нумерованные буферы работают с командой точка особым образом: вместо вставки содержимого буфера "1 эта команда вставляет содержимое следующего буфера ("2). Следующая команда u и команда точка (.) заменят в тексте содержимое буфера "2 содержимым буфера "3 и так далее, проходя через все девять буферов.

Чтение и запись файлов

Команда Выйти (ZZ)

Если в командной строке, вызывающей редактор vim, указывается имя файла, то редактор читает файл с диска в рабочий буфер. Затем команда ZZ, которая завершает сеанс редактирования, записывает содержимое рабочего буфера обратно в файл на диске. В этом разделе рассматриваются другие способы чтения текста в рабочий буфер и записи его в файл.

Чтение файлов

Команда Прочитать (:r)

Команда Прочитать считывает файл в рабочий буфер. Содержимое нового файла не переписывает текст в рабочем буфере, а размещается сразу же за единственным, указанным вами адресом (или за текущей строкой, если адрес не указан). Для записи файла в начало рабочего буфера можно использовать адрес 0. В команде Прочитать используется следующий синтаксис:

```
:[адрес]r [имя_файла]
```

Эта команда, как и все остальные, в которых используется двоеточие, при вводе этого символа появляется в строке состояния. В качестве *имени_файла* используется путевое имя файла, содержимое которого нужно прочитать, и должно быть завершено нажатием клавиши ВВОД. Если опустить *имя_файла*, vim прочитает с диска редактируемый вами файл.

Запись файлов

Команда Записать (:w)

Команда Записать записывает часть рабочего буфера или весь буфер в файл. Для записи части рабочего буфера можно указать адрес, а для определения файла, принимающего текст, можно указать его имя. Если не указать адрес или имя файла, vim записывает все содержимое рабочего буфера в редактируемый файл, обновляя тем самым файл на диске.

В течение длительного сеанса редактирования было бы неплохо время от времени пользоваться командой Записать. Если позже возникнет какая-нибудь проблема, то последняя копия рабочего буфера на диске останется в целости и сохранности. Если для выхода из vim воспользоваться командой :q!, то файл на диске будет точной копией той версии рабочего буфера, которая была на момент самого последнего вызова команды Записать.

У команды Записать есть два формата:

```
:[адрес]w[!] [имя_файла]
:[адрес]w>> имя_файла
```

Второй формат добавляет текст к уже существующему файлу. В *адресе* указывается та часть рабочего буфера, которую vim запишет в файл. *Адрес* придерживается той же самой формы, что и адрес, который используется командой Подставить (см. раздел «Адрес подстановки» на с. 196). Если *адрес* не указать, vim записывает все содержимое рабочего буфера. Необязательный параметр *имя_файла* является путевым именем файла, в который ведется запись. Если не указать *имя_файла*, то vim ведет запись в редактируемый файл.

Команда w! Поскольку команда Записать может очень быстро уничтожить большой объем работы, vim требует, чтобы за командой w в качестве защитного средства от случайной перезаписи файла вводился восклицательный знак (!). Восклицательный знак не нужен только при записи всего содержимого рабочего буфера в редактируемый файл (без использования *адреса* и *имени_файла*) и при записи части рабочего буфера или всего его содержимого в новый файл. А при записи части файла в редактируемый файл или при перезаписи другого файла нужно использовать восклицательный знак.

Идентификация текущего файла

Команда Файл (:f) предоставляет ту же самую информацию, что и команда Статус (CONTROL+G; см. раздел «Команда Статус» на с. 199). Имя файла, которое отображается с помощью команды Файл и является тем самым *именем_файла*, которое команда Записать использует в том случае, если команду :w ввести без *имени_файла*.

Установка параметров

Настройка редактора vim под ваши нужды и привычки производится путем установки параметров. Благодаря установке параметров выполняются такие функции, как отображение номеров строк, автоматическая вставка символов, генерируемых клавишей ВВОД, и включение поиска с последовательным уточнением в нестандартных режимах.

Установить параметры можно несколькими способами. Например, их можно установить для создания среды окружения для текущего сеанса редактирования в процессе использования vim. Или же параметры можно установить в ваших файлах запуска оболочки ~/.bash_profile (для оболочки bash), или ~/.tcshrc (для оболочки tcsh), или в файле запуска редактора vim — ~/.vimrc. Когда параметры установлены в любом из этих файлов, среда, настроенная под ваши запросы, будет доступна при каждом запуске vim, и вы сможете сразу же приступить к редактированию.

Установка параметров непосредственно из редактора vim

Для установки параметра в процессе использования редактора vim нужно ввести двоеточие (:), слово set, пробел и параметр (см. раздел «Параметры» на с. 205). Команда, по мере ее ввода, появится в строке состояния и будет выполнена после нажатия клавиши ВВОД. Следующая команда устанавливает режим поиска с последовательным уточнением для текущего сеанса редактирования:

```
:set incsearch
```

Установка параметров в файле запуска

Команда VIMINIT

При работе с оболочкой bash в ваш файл запуска ~/.bash_profile (см. главу 8, раздел «Файл .bash_profile» на с. 285) можно вставить строку, имеющую следующий формат:

```
export VIMINIT='set параметр1 параметр2 ...'
```

Элементы *параметр1* и *параметр2* следует заменить параметрами, выбранными из табл. 6.7. Элемент VIMINIT является переменной оболочки, значение которой считывается редактором vim. Следующая инструкция заставляет vim игнорировать регистр символов при поиске, отображать номера строк, использовать оболочку TC Shell для выполнения команд Linux и автоматически переносить текст за 15 символов от края экрана:

```
export VIMINIT='set ignorecase number shell=/bin/tcsh wrapmargin=15'
```

Если используется параметр сокращенной формы — abbreviations, эта строка имеет следующий вид:

```
export VIMINIT='set ic nu sh=/bin/tcsh wm=15'
```

При работе с оболочкой tcsh в ваш файл запуска ~/.tcshrc (см. раздел «Файлы запуска» на с. 369) нужно поместить строку следующего формата:

```
setenv VIMINIT 'set параметр1 параметр2 ...'
```

Здесь также нужно заменить *параметр1* и *параметр2* параметрами из табл. 6.7. Значения, помещаемые в одинарные кавычки, аналогичны тем, что были показаны в предыдущих примерах.

Файл запуска .vimrc

Вместо установки параметров редактора vim в файле запуска оболочки можно создать файл ~/.vimrc в вашем личном каталоге и установить параметры в этом файле. Создание файла .vimrc заставляет vim начинать работу с отключенным параметром compatible (см. раздел «Параметр compatible» на с. 177). Строки в файле .vimrc должны соответствовать следующему формату:

```
set параметр1 параметр2 ...
```

Далее приводятся примеры файлов .vimrc, осуществляющих те же функции, что и установка значения рассмотренной ранее переменной VIMINIT:

```
$ cat ~/.vimrc
set ignorecase
set number
set shell=/bin/tcsh
set wrapmargin=15
$ cat ~/.vimrc
set ic nu sh=/bin/tcsh wm=15
```

Параметры, устанавливаемые с помощью значения переменной VIMINIT, имеют приоритет над теми, которые установлены в файле .vimrc.

Параметры

В табл. 6.7 приводится список некоторых наиболее полезных параметров vim. Если ввести команду `:set ВВОД`, редактор vim выведет весь список параметров и покажет их текущие установки. Команда `:set ВВОД` отображает список только тех параметров, для которых установлены значения, отличающиеся от их значений по умолчанию. Параметры делятся на два класса: те, которые содержат знак равенства (и воспринимают значения), и те, у которых может присутствовать префикс `no` (переключатель, позволяющий включать и выключать параметр). Смысл параметра-переключателя можно изменить путем ввода команды `:set [no]`*параметр*. Например, для включения или выключения нумерации строк следует ввести команду `:set number` (или `:set nonumber`). Для изменения значения параметра, который воспринимает какую-нибудь величину (и использует знак равенства), следует ввести команду вида `:set shiftwidth=15`.

У многих параметров есть сокращенные формы — например, `nu` для `number`, `nonu` для `nonumber` и `sw` для `shiftwidth`. Сокращения приводятся в левом столбце табл. 6.7 сразу же за именем параметра.

Таблица 6.7. Параметры

Параметр	Эффект
Разрешение применения специальных символов в строках поиска `Magic`	См. раздел «Использование специальных символов в строках поиска» на с. 194. По умолчанию следующие символы имеют в строке поиска специальные значения: . [ ] * При установке параметра `nomagic` эти символы лишаются специального значения. Установка параметра `magic` возвращает им специальное значение. Символы ^ и $ в строке поиска всегда имеют специальное значение, независимо от установки данного параметра
Автоматический отступ `autoindent, ai`	Свойство автоматического отступа работает с параметром `shiftwidth` для обеспечения обычного набора отступов для программ или табличного материала. По умолчанию это свойство отключено. Его можно включить, установив параметр `autoindent`, и отключить, установив параметр `noautoindent`. Когда автоматический отступ включен и редактор vim находится в режиме ввода, для перемещения курсора от левой границы (или отступа) к следующей позиции отступа следует воспользоваться комбинацией CONTROL+T, а нажатие клавиши ВВОД приведет к перемещению курсора к левому краю следующей строки, находящемуся под первым символом предыдущей строки, а использование комбинации CONTROL+D возвращает курсор назад на один отступ. Комбинации CONTROL+T и CONTROL+D работают только до размещения текста на строке
Автоматическая запись `autowrite, aw`	Если вы не задали операцию явным образом (как при вводе команды `:n` на редактирование следующего файла), то по умолчанию vim перед записью рабочего буфера выводит запрос на запись. Параметр `autowrite` заставляет vim записывать рабочий буфер при использовании таких команд, как `:n`, вызывающих редактирование другого файла, автоматически. Это свойство можно отключить, установив параметр `noautowrite` или `noaw`
Подача светового сигнала `flash, fl`	Обычно редактор vim при работе в командном режиме заставляет терминал при вводе неверной команды или нажатии клавиши ESCAPE подавать звуковой сигнал. Установка параметра `flash` заставляет терминал подавать не звуковой, а световой сигнал. Установка параметра `noflash` заставляет вернуться к подаче звукового сигнала. Этот параметр поддерживается не всеми терминалами и эмуляторами терминалов

продолжение ⇩

Таблица 6.7 (*продолжение*)

Параметр	Эффект
Игнорирование регистра при проведении поисков ignorecase, ic	Обычно редактор vim проводит поиски с учетом регистра символов, различая буквы верхнего и нижнего регистров. Но после установки параметра ignorecase он проводит поиски без учета регистра символов. Для возвращения чувствительности к регистру следует установить параметр noignorecase
Поиск с последовательным уточнением incsearch, is	См. раздел «Сравнение обычного поиска и поиска с последовательным уточнением» на с. 194. По умолчанию vim не ведет поиск с последовательным уточнением. Чтобы заставить vim вести такой поиск, нужно установить параметр incsearch. А чтобы заставить vim не проводить поиск с последовательным уточнением, нужно установить параметр noincsearch
Отображение невидимых символов List	Чтобы заставить vim отображать каждый символ табуляции в виде ^I и помечать конец каждой строки символом $, нужно установить параметр list. Чтобы отображать символы табуляции в виде пробелов и не помечать концы строк, нужно установить параметр nolist
Отображение номеров строк number, nu	Обычно редактор vim не отображает номеров, связанных с каждой строкой. Для отображения номеров строк нужно установить параметр number. Чтобы отменить нумерацию строк, нужно установить параметр nonumber. Номера строк не являются частью файла и не хранятся вместе с файлом, а также не отображаются при выводе файла на печать. Они появляются на экране только при работе с vim
Автоматический перенос строк Wrap	Автоматический перенос строк управляет тем, как vim отображает строки, не помещающиеся на экране. Чтобы заставить vim переносить длинные строки и продолжать их на новой строке, нужно установить параметр wrap (он устанавливается по умолчанию). Если установить параметр nowrap, vim будет проводить усечение длинных строк по правой границе экрана
Граница автоматического переноса строк wrapmargin=*nn*, wm=*nn*	Граница автоматического переноса строк заставляет vim разбивать вставляемый текст при достижении примерно того количества символов, оставшегося до правого края экрана, которое указано при установке параметра. Редактор vim разбивает текст, вставляя разделитель строк на ближайшую границу слова, отделенного пробелом. Установка границы автоматического переноса строк удобна в том случае, когда нужно, чтобы все строки были примерно одинаковой длины. Это свойство избавляет вас от необходимости помнить о нажатии клавиши ВВОД в конце каждой введенной строки. При установке параметра wrapmargin=*nn* под *nn* понимается количество символов от правого края экрана, при достижении границы которых vim разбивает текст. Это количество не является шириной столбца текста, а представляет собой расстояние от конца текста до правого края экрана. Установка нулевого значения для границы автоматического переноса строк приводит к отключению этого параметра. По умолчанию граница автоматического переноса строк отключена (установлена в нуль)
Вывод отчета report=*nn*	Этот параметр заставляет vim выводить в строку состояния отчет при внесении изменений, касающихся как минимум *nn* строк. Например, если параметр report имеет значение 7 и вы удалили семь строк, vim выводит сообщение «7 lines deleted» («Удалено 7 строк»). При удалении шести и менее строк vim не выводит никакого сообщения. По умолчанию для report установлено значение 5

Параметр	Эффект
Включение прокрутки scroll=*nn*, scr=*nn*	Этот параметр управляет количеством строк, прокручиваемых на экране с помощью комбинаций CONTROL+D и CONTROL+U (см. раздел «Команды CONTROL+D и CONTROL+U» на с. 187). По умолчанию параметр scroll настроен на значение, равное половине высоты окна. Чтобы изменить значение прокрутки, существует два способа. Можно ввести количество строк перед использованием комбинации CONTROL+D или CONTROL+U, и vim установит для scroll это значение. А можно установить значение scroll в явном виде, с помощью команды scroll=*nn*, где *nn* — это необходимое количество строк прокрутки при каждом вводе команды CONTROL+D или команды CONTROL+U
Указание оболочки shell=*путевое_имя*, sh=*путевое_имя*	При работе с vim можно заставить его вызвать новый экземпляр оболочки. Вы можете либо создать копию интерактивной оболочки (если нужно запустить несколько команд) или запустить одну команду. Параметр shell определяет, какую именно оболочку будет вызывать vim. По умолчанию vim устанавливает параметр shell на ту оболочку, которая вызывалась при входе в систему. Чтобы поменять оболочку, установите параметр shell=*путевое_имя*, где *путевое_имя* является абсолютным путевым именем файла нужной вам оболочки
Величина отступа shiftwidth=*nn*, sw=*nn*	Этот параметр управляет функционированием комбинаций CONTROL+T и CONTROL+D в режиме ввода (см. строку «Автоматический отступ» в данной таблице). При установке параметра shiftwidth=*nn* под *nn* подразумевается количество позиций отступа (по умолчанию оно равно 8). Установка величины отступа аналогична установке позиций табуляции на пишущей машинке, но в случае с величиной отступа дистанция между позициями табуляции остается постоянной
Показ соответствия showmatch, sm	Этот параметр пригодится тем программистам, которые работают с языками, использующими фигурные ({}) или круглые скобки в качестве разделителей выражений (Lisp, C, Tcl и т. д.). Когда параметр showmatch установлен и вы вводите код (в режиме ввода) и набираете закрывающую фигурную или круглую скобку, курсор на мгновение перепрыгивает на соответствующую ей открывающую фигурную или круглую скобку (то есть на соответствующий элемент на том же уровне вложенности). После обозначения соответствующего элемента курсор возвращается на свою прежнюю позицию. При вводе правой фигурной или круглой скобки, не имеющей соответствия, vim подает звуковой сигнал. Чтобы выключить автоматический показ соответствия, нужно установить параметр noshowmatch
Отображение режима showmode, smd	Чтобы vim в режиме ввода отображал этот режим в нижнем правом углу экрана, установите параметр showmode (он устанавливается по умолчанию). Чтобы vim не отображал режим, установите параметр noshowmode
Включение совместимости с редактором vi compatible, cp	См. раздел «Параметр compatible» на с. 177. По умолчанию vim не поддерживает совместимость с редактором vi. Чтобы заставить vim поддерживать совместимость с vi, нужно установить параметр compatible. Чтобы заставить vim отключить совместимость с vi, нужно установить параметр nocompatible
Продолжение сканирования wrapscan, ws	По умолчанию, когда в процессе поиска следующего появления искомого элемента будет достигнут конец рабочего буфера, vim продолжает поиск с начала рабочего буфера. Обратное утверждение имеет силу для поиска предыдущего появления искомого элемента. Установка параметра nowrapscan останавливает поиск по достижении обеих границ рабочего буфера. Чтобы вести поиск с продолжением сканирования при достижении границ рабочего буфера, нужно установить параметр wrapscan

Расширенные технологии редактирования

В этом разделе представлен ряд команд, которые могут пригодиться после освоения работы с vim.

Использование маркеров

Для более удобной адресации при работе с vim можно устанавливать и использовать маркеры. Маркер устанавливается командой `mc`, где `c` — это любой символ. (Лучше использовать буквы, поскольку некоторые символы, например одинарная кавычка, имеют при использовании в качестве маркеров особый смысл.) При выходе из vim маркеры не сохраняются.

После установки маркера его можно использовать точно так же, как номер строки. Можно перемещать курсор в начало той строки, на которой имеется маркер, для чего нужно перед именем маркера поставить одинарную кавычку. Например, для установки маркера `t` нужно установить курсор на нужную строку и ввести команду `mt`. В течение всего текущего сеанса редактирования, пока этот маркер не будет снят или строка, которая им помечена, не будет удалена, можно будет вернуться к началу строки, на которой установлен этот маркер, путем ввода команды `'t`.

Если ввести следующую команду, то можно удалить весь текст от текущей строки и до той строки, на которой установлен маркер `r`:

```
d'r
```

Для перехода именно к той позиции в строке, на которой установлен маркер, можно воспользоваться обратным апострофом (`, который также называется перевернутой одинарной кавычкой). После установки маркера `t` можно переместить курсор на место установки этого маркера (а не к началу той строки, на которой он установлен), воспользовавшись командой `t. Следующая команда удаляет весь текст от текущей строки и до символа, на который был установлен маркер `r; остальная часть строки, на которой установлен маркер, остается нетронутой:

```
d`r
```

Маркеры можно использовать в адресах команд вместо номеров строк. Следующая команда заменяет все встреченные строковые значения `The` на `THE` на всех строках от маркера `m` и до текущей строки (маркер `m` должен быть установлен до текущей строки):

```
:'m,.s/The/THE/g
```

Редактирование других файлов

Следующая команда заставляет vim редактировать файл, имя которого в ней указано:

`:e[!] [`*`имя_файла`*`]`

Если нужно сохранить содержимое рабочего буфера, то перед тем как вводить эту команду, его нужно записать (используя команду `:w`). Если содержимое рабочего буфера сохранять не нужно, vim требует воспользоваться восклицательным знаком, чтобы дать ему понять, что вы намерены потерять всю работу, которая была проделана с момента последнего сохранения содержимого рабочего буфера. Если этой команде не предоставить *имя_файла*, vim откроет для редактирования тот файл, с которым вы работали.

Команда :e! Команда `:e!` начинает сеанс редактирования заново. Эта команда возвращает рабочий буфер к тому состоянию, которое он имел к моменту его последней записи, или, если вы не вели такую запись, к тому состоянию, в котором он был в самом начале редактирования файла. Эта команда пригодится в том случае, когда допущена ошибка при редактировании файла и принято решение, что будет проще не исправлять ошибку, а начать все сначала. Поскольку команда `:e` не удаляет содержимое универсального буфера или именованных буферов, можно сохранить текст из одного файла в буфере, воспользоваться командой `:e` для редактирования другого файла и вставить текст из буфера в этот файл.

Команда :e# Команда `:e#` закрывает текущий файл и открывает последний файл, который редактировался, помещая курсор на ту строку, на которой он стоял при последнем закрытии файла. Если перед вводом этой команды не сохранить редактируемый файл, то vim предложит

сделать это. Если установить параметр автоматической записи — autowrite (см. табл. 6.7), то это все равно не остановит vim от выдачи такого предложения.

Команды :n и :rew

Команда :e# может помочь скопировать блоки текста из одного файла в другой. Когда при вызове vim в качестве аргументов используются имена нескольких файлов, вы можете использовать команду :n для редактирования следующего файла, команду :e# для редактирования только что редактировавшегося файла и команду :rew для перехода в начало последовательности файлов для повторного редактирования самого первого файла. При переходе от файла к файлу можно копировать текст из одного файла в буфер и вставлять этот текст в другой файл. Чтобы заставить vim перед открытием следующего файла закрывать текущий файл без записи изменений, следует воспользоваться командой :n!.

Макрокоманды и сокращения

Команда :map

Редактор vim позволяет создавать макрокоманды и сокращения. Команда :map определяет клавишу или последовательность клавиш, которая совершает некие действия в командном режиме. Следующая команда назначит комбинации CONTROL+X выполнение команд, которые будут искать следующую левую квадратную скобку в текущей строке (f[), удалять все символы от этой скобки до следующей правой квадратной скобки (df]) на той же строке, удалять следующий символ (x), перемещать курсор вниз на две строки (2j) и последним действием перемещать курсор к началу строки (0):

```
:map ^X f[df]x2j0
```

Можно, конечно, использовать клавиши ESCAPE- и CONTROL-последовательностей, но лучше этого не делать, чтобы избежать переназначения тех клавиш и последовательностей, которые являются vim-командами. Чтобы вывести список текущих назначений, введите команду :map без аргументов. Чтобы процитировать ряд символов, которые нужно ввести в строку команды :map, может пригодиться комбинация CONTROL+V (см. раздел «Команда CONTROL+V» на с. 189).

Команда :abbrev

Команда :abbrev похожа на команду :map, но создает аббревиатуры, которые можно использовать в режиме ввода. Когда вы работаете в режиме ввода и набираете строку, определенную в команде :abbrev, за которой нажимаете ПРОБЕЛ, редактор vim заменяет эту строку и пробел символами, указанными при определении строки. Чтобы упростить использование этой функции, при создании аббревиатур следует избегать распространенных предложений или групп символов. Следующая команда определяет группу символов ZZ в качестве аббревиатуры для Sam the Great:

```
:abbrev ZZ Sam the Great
```

Хотя сочетание ZZ является vim-командой, оно в этом качестве используется только в командном режиме. А в режиме ввода при использовании в качестве аббревиатуры оно специального значения не имеет.

Выполнение команд оболочки из редактора vim

Команда :sh

В процессе работы с vim команды оболочки можно выполнить несколькими способами. Например, можно с помощью ввода следующей команды и нажатия клавиши ВВОД запустить новый экземпляр интерактивной оболочки:

```
:sh
```

Используемый в редакторе vim параметр shell (см. табл. 6.7) определяет, экземпляр какой именно оболочки будет запущен (обычно используется оболочка bash или tcsh). По умолчанию будет запущена та же оболочка, которая использовалась при входе в систему.

После завершения работы с оболочкой можно вернуться в vim путем выхода из оболочки (для чего следует воспользоваться комбинацией CONTROL+D или ввести команду exit).

ЕСЛИ :SH РАБОТАЕТ НЕПРАВИЛЬНО

В зависимости от того, как была сконфигурирована оболочка, команда:sh может вести себя немного странно. Она может выводить предупреждения или даже зависнуть. Проэкспериментируйте с командой :sh, чтобы убедиться в том, что она при имеющихся настройках работает правильно. Если возникают странности в ее работе, то перед вызовом :sh может потребоваться перенастроить vim-параметр shell на использование другой оболочки. Например, следующая команда заставляет vim использовать с командой :sh оболочку tcsh:

```
:set shell=/bin/tcsh
```

Чтобы после запуска команды :sh отображалась правильная оболочка, может потребоваться изменение значения переменной среды окружения SHELL.

РЕДАКТИРУЙТЕ ТОЛЬКО ОДНУ КОПИЮ ФАЙЛА

При запуске нового экземпляра оболочки с помощью команды :sh следует помнить, что вы все еще используете редактор vim. Довольно часто допускается ошибка, связанная с попыткой редактирования того же самого файла из новой оболочки. Не упускайте из виду то обстоятельство, что vim уже редактирует этот файл. Поскольку при редактировании одного и того же файла из двух экземпляров редактора может быть утеряна информация, подобная ошибка с вашей стороны вызывает предупреждение со стороны редактора vim. Пример сообщения, выводимого vim, показан в разделе «Блокировки файлов» на с. 181.

Команда :!команда

Можно выполнить командную строку оболочки из самого редактора vim путем ввода следующей команды, заменив при этом элемент *команда* командной строкой, которую нужно выполнить, и завершив эту команду нажатием клавиши ВВОД:

```
:!команда
```

Для выполнения этой команды редактор vim запустит новый экземпляр оболочки. Когда команда доработает до конца, только что запущенная оболочка вернет управление редактору.

Команда :!!команда

Можно выполнить команду из vim и одновременно заставить ее заменить текущую строку выходной информацией этой команды. Если не нужно заменять какой-нибудь текст, можно перед вводом следующей команды установить курсор на пустую строку:

```
!!команда
```

При вводе первого восклицательного знака ничего не происходит. А при вводе второго такого же знака vim перемещает курсор в строку состояния и дает возможность ввести команду, которую нужно выполнить. Поскольку эта команда переводит vim в режим последней строки, вы должны завершить ввод команды нажатием клавиши ВВОД (точно так же, как и при завершении ввода большинства команд оболочки).

Можно также выполнить команду из vim, перенаправив стандартный ввод в команду на получение информации из всего редактируемого файла или из его части и перенаправив стандартный вывод команды на вход редактируемого файла. Команды такого рода удобно применять для сортировки списка прямо внутри файла.

Для определения блока текста, который станет стандартным вводом для команды, нужно переместить курсор к одной из границ блока текста, затем ввести восклицательный знак, за которым ввести команду, которая обычно перемещает курсор на другую границу блока текста. Например, если курсор находится в начале файла и вам нужно указать на весь файл, введите команду `!G`. Если нужно указать на часть файла, которая

находится между позицией курсора и маркером b, следует ввести команду !'b. После того как введена команда перемещения курсора, vim выводит восклицательный знак в строке состояния и ждет от вас ввода команды оболочки.

Чтобы отсортировать имеющийся в файле список имен, переместите курсор к началу списка и установите в этом месте маркер q, воспользовавшись командой mq. Затем переместите курсор к концу списка и введите следующую команду:

```
!'qsort
```

Нажмите клавишу ВВОД и ждите. Через несколько секунд отсортированный список заменит прежний список на экране. Если команда не поведет себя ожидаемым образом, можно, как обычно, отменить изменения, воспользовавшись командой u. Дополнительные сведения о сортировке даны в части V в разделе «sort» на с. 820.

КОМАНДА ! МОЖЕТ УНИЧТОЖИТЬ ФАЙЛ

Если ввести неправильную команду или ошибиться при наборе команды, можно уничтожить файл (например, если команда подвесит vim или остановит его работу). Поэтому рекомендуем сохранять ваш файл перед вводом такой команды. Спасти ситуацию может команда Отменить (см. раздел «Команды Отменить (u и U)» на с. 189). Команда :e! (см. раздел «Редактирование других файлов» на с. 208) способна избавить от изменений, возвращая буфер к состоянию его последнего сохранения.

Как и в случае команды :sh, оболочка, используемая по умолчанию, может работать с командой ! неправильно. Перед применением этой команды в реальной обстановке может потребоваться тестирование оболочки. Если оболочка, используемая по умолчанию, не работает должным образом, нужно изменить значение параметра shell.

Единицы отсчета

Многие vim-команды работают с блоками текста в диапазоне от одного символа до нескольких абзацев. Размер блока текста задается в единицах отсчета. Перед количеством повторений можно указать единицу отсчета исходя из широкого набора таких единиц (см. раздел «Количество повторений» на с. 214). В этом разделе даются определения различных единиц отсчета.

Символ

Эта единица отсчета представляет собой один символ, независимо от того, отображаемый он или нет, включая пробелы и символы табуляции. В качестве примера можно привести следующие символы:

```
a q A . 5 R - > TAB ПРОБЕЛ
```

Слово

Единица отсчета «слово» соотносится со словом английского языка, строковым значением из одного или нескольких символов, ограниченным с обеих сторон любой комбинацией из одного или нескольких следующих элементов: знаков пунктуации, пробела, символа табуляции, цифровых символов или разделителей строк. Кроме этого, vim рассматривает каждую группу знаков пунктуации в качестве отдельного слова (табл. 6.8).

Таблица 6.8. Слова

Количество слов	Текст
1	Pear
2	pear!
2	pear!)
3	pear!) The
4	pear!) “The
11	This is a short, concise line (no frills) *Это короткая лаконичная строка (без излишеств)*

Слово, отделенное пробелами

Слова, отделенные пробелами, — это те же самые слова, но включающие примыкающие к ним знаки пунктуации. Для слов, отделенных пробелами, разделителями считаются следующие элементы: пробел, символ табуляции или разделитель строк (табл. 6.9).

Таблица 6.9. Слова, отделенные пробелами

Количество слов	Текст
1	Pear
1	Pear!
1	Pear!)
2	Pear!) The
2	Pear!) “The
8	This is a short, concise line (no frills) *Это короткая лаконичная строка (без излишеств)*

Строка

Строка — это череда символов, которая ограничивается разделителями строк и отображается необязательно на одной физической строке экрана. Можно ввести очень длинную (логическую) строку, которая будет несколько раз перенесена (продолжена на следующей физической строке) или скроется за правой границей экрана. Лучше избегать создания длинных логических строк, в идеале нужно завершать строки нажатием клавиши ВВОД еще до того, как они достигнут правого края экрана. Такое завершение строк гарантирует наличие на каждой физической строке одной логической строки и позволяет избежать неприятностей при редактировании и форматировании текста.

Некоторые команды с логическими строками, которые оказываются длиннее ширины экрана, должным образом не работают. Например, если курсор находится на длинной логической строке, которая несколько раз была перенесена на несколько физических строк, то одно нажатие клавиши ВВОД приводит к перемещению курсора более чем на одну строку. Для того чтобы разбить длинные логические строки на несколько более коротких, можно воспользоваться командой fmt (см. часть V, раздел «fmt» на с. 708).

Предложение

Предложением считается английское предложение или его эквивалент. Предложение начинается с конца предыдущего предложения и заканчивается точкой, восклицательным или вопросительным знаком, за которыми следуют два пробела или разделитель строк (табл. 6.10).

Таблица 6.10. Предложения

Количество предложений	Текст
Одно: после первой точки стоит только один пробел, а после второй точки стоит разделитель строк	That's it. This is one sentence *Совершенно верно. Это одно предложение*
Два: после первой точки стоят два пробела, а после второй точки стоит разделитель строк	That's it. This is two sentence *Совершенно верно. Это два предложения*
Три: после первых двух вопросительных знаков стоят по два пробела, а после восклицательного знака стоит разделитель строк	What? Three sentences? One line! *Что? Три предложения? Одна строка!*
Одно: разделитель строк стоит после точки	This sentence takes *Это предложение* up a total of *занимает целых* three lines *три строки*

Абзац

Абзацу предшествует и за ним следует одна или несколько пустых строк. Пустая строка состоит из двух следующих друг за другом разделителей строк (табл. 6.11).

Таблица 6.11. Абзацы

Количество абзацев	Текст
Один: пустая строка перед и после текста	One paragraph *Один абзац*
Один: пустая строка перед и после текста	This may appear to be more than one paragraph. *Может показаться, что здесь более одного абзаца.* Just because there are two indentions does not mean it qualifies as two paragraphs *Наличие двух отступов не означает, что этот текст должен считаться двумя абзацами*
Три: три блока текста, разделенные пустыми строками	Even though in English this is only one sentence, *Даже при том, что с точки зрения английского языка это всего лишь одно предложение,* vim considers it to be three paragraphs *vim считает, что это три абзаца*

Экран (окно)

При работе с vim экран или окно эмулятора терминала может отображать одно или несколько логических окон информации. Окно отображает все содержимое рабочего буфера или его часть. На рис. 6.5 показан экран с двумя окнами.

Количество повторений

Число, которое ставится перед единицей отсчета (см. раздел «Единицы отсчета» на с. 211), обозначает количество повторений. Так же как цифра 5 в 5 дюймах заставляет вас рассматривать 5 дюймов в качестве отдельной единицы отсчета, так и количество повторений заставляет vim создавать группу из нескольких единиц отсчета. Например, команда w перемещает курсор вперед на одно слово, команда 5w перемещает его вперед на 5 слов, а команда 250w перемещает его вперед на 250 слов. Если не указывать количество повторений, vim предположит, что оно равно 1. Если количество повторений должно переместить курсор за границу файла, то курсор останется в конце этого файла.

Заключение

В заключительный раздел этой главы, посвященной vim, включены все команды, рассмотренные в данной главе, а также некоторые другие команды. В табл. 6.12 приведен список способов вызова редактора vim из командной строки.

Таблица 6.12. Вызов редактора vim

Команда	Результат
`vim` *имя_файла*	Редактирует файл с именем *имя_файла*, начиная с первой строки
`vim +`*n имя_файла*	Редактирует файл с именем *имя_файла*, начиная со строки *n*
`vim +` *имя_файла*	Редактирует файл с именем *имя_файла*, начиная с последней строки
`vim +/`*образец имя_файла*	Редактирует файл с именем *имя_файла*, начиная с первой строки, содержащей *образец*
`vim -r` *имя_файла*	Восстанавливает файл с именем *имя_файла* после системного сбоя
`vim -R` *имя_файла*	Редактирует файл с именем *имя_файла* в режиме только для чтения (то же самое, что и открытие файла с помощью команды `view`)

Чтобы воспользоваться командами перемещения курсора по единицам отсчета (табл. 6.13), нужно находиться в командном режиме. Эти единицы отсчета можно использовать с командами Изменить, Удалить и Выбрать. Перед каждой из этих команд может быть указано количество повторений.

Таблица 6.13. Перемещение курсора по единицам отсчета

Команда	Перемещение курсора
ПРОБЕЛ, l (буква «l») или СТРЕЛКА ВПРАВО	На позицию вправо
h или СТРЕЛКА ВЛЕВО	На позицию влево
w	На слово вправо
W	На слово, отделенное пробелом вправо
B	На слово влево
B	На слово, отделенное пробелом влево
$	В конец строки
E	В конец слова вправо
E	В конец слова, отделенного пробелом вправо

Команда	Перемещение курсора
0 (нуль)	В начало строки (не может использоваться с количеством повторений)
ВВОД	В начало следующей строки
j или СТРЕЛКА ВНИЗ	Вниз на одну строку
-	В начало предыдущей строки
k или СТРЕЛКА ВВЕРХ	Вверх на одну строку
)	В конец предложения
(	В начало предложения
}	В конец абзаца
{	В начало абзаца
%	Перемещение к соответствующей скобке того же типа и на том же уровне вложенности

В табл. 6.14 показаны команды, позволяющие просматривать различные части рабочего буфера.

Таблица 6.14. Просмотр рабочего буфера

Команда	Перемещение курсора
CONTROL+D	Вперед на половину окна
CONTROL+U	Назад на половину окна
CONTROL+F или PAGE DOWN	Вперед на одно окно
CONTROL+B или PAGE UP	Назад на одно окно
*n*G	На строку *n* (без *n* — на последнюю строку)
H	В верхнюю часть окна
M	На середину окна
L	В нижнюю часть окна

Команды в табл. 6.15 позволяют добавлять текст к буферу. Все эти команды, за исключением r, оставляют vim в режиме ввода. Для возврата в командный режим нужно нажать клавишу ESCAPE.

Таблица 6.15. Добавление текста

Команда	Место добавления текста
I	Перед курсором
I	Перед первым непробельным символом в строке
A	После курсора
A	В конце строки
O	Открывает строку под текущей строкой
O	Открывает строку над текущей строкой
R	Заменяет текущий символ (клавишу ESCAPE использовать не нужно)
R	Заменяет символы, начиная с текущего (перезаписывает символы, пока не будет нажата клавиша ESCAPE)

В табл. 6.16 приведен список команд, удаляющих и изменяющих текст. В этой таблице *M* — это единица отсчета, перед которой можно поставить количество повторений, *n* — это необязательное количество повторений, а *c* — любой символ.

Таблица 6.16. Удаление и изменение текста

Команда	Результат
*N*x	Удаляет количество символов, указанное в *n*, начиная с текущего символа
*n*X	Удаляет *n* символов перед текущим символом, начиная с символа, предшествующего текущему
d*M*	Удаляет текст, определяемый *M*
*N*dd	Удаляет *n* строк
Dt*c*	Удаляет следующий символ *c* на текущей строке
D	Удаляет текст до конца строки
n~	Изменяет регистр следующих *n* символов
Следующие команды оставляют vim в режиме ввода. Для возвращения в командный режим нужно нажать клавишу ESCAPE	
*N*s	Подставляет *n* символов
S	Осуществляет подстановку вместо всей строки
c*M*	Изменяет текст, определяемый *M*
*N*cc	Изменяет *n* строк
Ct*c*	Изменяет следующий символ *c* в текущей строке
C	Изменяет все до конца строки

В табл. 6.17 приведен список команд поиска. Здесь *rexp* означает регулярное выражение, которое может быть простой строкой символов.

Таблица 6.17. Ведение поиска

Команда	Результат
/*rexp*ВВОД	Поиск *rexp* вперед
?*rexp*ВВОД	Поиск *rexp* назад
N	Точное повторение исходного поиска
N	Точное повторение исходного поиска в противоположном направлении
/ВВОД	Повторение исходного поиска вперед
?ВВОД	Повторение исходного поиска назад
F*c*	Позиционирование курсора на следующий символ *c* в текущей строке
F*c*	Позиционирование курсора на предыдущий символ *c* в текущей строке
T*c*	Позиционирование курсора на символ перед следующим символом *c* (на символ слева от него)
T*c*	Позиционирование курсора на символ после следующего символа *c* (на символ справа от него)
;	Повторение последней команды f, F, t или T

Для команды Поставить используется следующий формат:

```
:[адрес]s/строка_поиска/строка_замены[/g]
```

где *адрес* — номер одной строки или номера двух строк, разделенные запятой. Символ точки (.) обозначает текущую строку, символ $ обозначает последнюю строку, а символ % — весь файл. Вместо номера строки можно использовать маркер или строку поиска. Элемент *строка_поиска* представляет собой регулярное выражение, которое может быть простой строкой символов. Элемент *строка_замены* — это строка, которая будет подставлена. Ключ *g* служит признаком глобальной замены (более одной замены на строку).

В табл. 6.18 приведен список vim-команд общего назначения.

Таблица 6.18. Команды общего назначения

Команда	Результат
J	Объединяет текущую и следующую строку
Точка (.)	Повторяет самую последнюю команду, внесшую в текст изменения
:w *имя_файла*	Записывает содержимое рабочего буфера в файл с именем *имя_файла* (или в текущий файл, если *имя_файла* не задано)
:q	Завершает работу vim
ZZ	Записывает содержимое рабочего буфера в текущий файл и завершает работу vim
:f или CONTROL+G	Отображает имя файла, состояние, номер текущей строки, количество строк в рабочем буфере и процент той части рабочего буфера, который предшествует текущей строке
CONTROL+V	Вставляет следующий символ в виде литерала, даже если он является vim-командой (используется в режиме ввода)

В табл. 6.19 приведен список команд, выбирающих текст и помещающих его в рабочий буфер. В этой таблице *M* — это единица отсчета, перед которой можно поставить количество повторений, а *n* — это количество повторений. Перед любой из этих команд можно поставить имя буфера, используя форму "x, где x — это имя буфера (буква в диапазоне a–z).

Таблица 6.19. Выборка и помещение текста в рабочий буфер

Команда	Результат
y*M*	Выбирает текст, определяемый с помощью *M*
*N*yy	Выбирает *n* строк
Y	Выбирает все до конца строки
P	Помещает текст перед или выше
P	Помещает текст за или ниже

В табл. 6.20 приведен список расширенных vim-команд.

Таблица 6.20. Расширенные команды

Команда	Результат
M*x*	Устанавливает маркер *x*, где *x* — буква от a до z
'' (две одинарные кавычки)	Перемещает курсор на его предыдущую позицию

продолжение ⇩

Таблица 6.20 (*продолжение*)

Команда	Результат
'x	Перемещает курсор на строку, на которой находится маркер *x*
`x	Перемещает курсор к символу, на котором установлен маркер *x*
:e *имя_файла*	Редактирует файл с именем *имя_файла*, требуя от вас перед его редактированием сохранить изменения, внесенные в текущий файл (с помощью команды :w или установленного параметра `autowrite`). Для отказа от изменений, внесенных в текущий файл, используется команда :e! *имя_файла*. Для отказа от изменений, внесенных в текущий файл, и начала редактирования его сохраненной версии используется команда :e! без указания имени файла
:n	Редактирует следующий файл, когда vim запущен с указанием в качестве аргументов нескольких имен файлов. Требует от вас записи изменений, внесенных в текущий файл (с помощью команды :w или установленного параметра `autowrite`) . Для отказа от изменений, внесенных в текущий файл, и перехода к редактированию следующего файла используется команда :n!
:rew	Вызывает переход к самому первому имени файла, когда vim запущен с указанием в качестве аргументов нескольких имен файлов и начинает редактирование первого файла. Перед началом редактирования первого файла требует от вас записи изменений, внесенных в текущий файл (с помощью команды :w или установленного параметра `autowrite`). Для отказа от изменений, внесенных в текущий файл, и перехода к редактированию первого файла используется команда :rew!
:sh	Запускает оболочку. Чтобы вернуться в vim, нужно выйти из оболочки
:!*команда*	Запускает оболочку и выполняет *команду*
!!*команда*	Запускает оболочку, выполняет *команду* и помещает ее выходную информацию в рабочий буфер, заменяя ею текущую строку

Упражнения

1. Как заставить vim перейти в режим ввода? Как вернуть vim обратно в командный режим?
2. Что такое рабочий буфер? Назовите два способа записи содержимого рабочего буфера на диск.
3. Предположим, что вы редактируете файл, который содержит следующий абзац, и курсор находится на второй тильде (~):

```
The vim editor has a command, tilde (~),
that changes lowercase letters to uppercase, and vice versa.
The ~ command works with a Unit of Measure or

a Repeat Factor, so you can change
the case of more than one character at a time.
```

```
У редактора vim есть команда
превращающая буквы верхнего
регистра в нижний, и наоборот

Команда ~ работает с единицей отсчета
или с количеством повторений,
позволяя изменить
регистр сразу нескольких символов.
```

Как можно:

1) переместить курсор к концу абзаца?
2) переместить курсор к началу слова Unit?
3) заменить слово `character` словом `letter`?

4. При работе в vim курсор находится на первой букве слова, а вы вводите команду x, за которой следует команда p. Объясните, что при этом произойдет.
5. Чем отличаются друг от друга следующие команды?
   1) i и I
   2) a и A
   3) o и O
   4) r и R
   5) u и U
6. Какой командой нужно воспользоваться для поиска в обратном направлении имеющихся в рабочем буфере строк, которые начинаются со слова it?
7. Какая команда используется для подстановки вместо всех появлений фразы this week фразы next week?
8. Рассмотрите следующий сценарий. Вы запускаете vim для редактирования существующего файла. Вы вносите в файл множество изменений, а затем понимаете, что удалили очень важный раздел файла в самом начале сеанса редактирования. Вам нужно вернуть этот раздел, но при этом не хочется потерять все остальные внесенные изменения. Что нужно сделать?
9. Как переместить текущую строку в самое начало файла?
10. Воспользуйтесь редактором vim для создания файла letter_e, состоящего из букв e (см. раздел «bzip2: Сжатие файла» на с. 80). Используйте для этого как можно меньше vim-команд. Какими vim-командами вы воспользуетесь?

Упражнения повышенной сложности

1. Какой командой следует воспользоваться для извлечения абзаца из одного файла и вставки его в другой файл?
2. Создайте файл, содержащий следующий список, а затем выполните команду из редактора vim для сортировки списка и отображения его в двух столбцах. (Подсказка: дополнительные сведения о команде pr даны в части V.)

```
Command mode
Input mode
Last Line mode
Work buffer
General-Purpose buffer
Named buffer
Regular Expression
Search String
Replacement String
Startup File
Repeat Factor
```

3. Чем именованные буферы отличаются от универсального буфера?
4. Предположим, что ваша версия vim не поддерживает многократное применение команды Отменить. Если вы удаляете строку текста, затем удаляете еще одну строку, а затем и третью строку, какой командой вы воспользуетесь для восстановления первых двух удаленных вами строк?
5. Какой командой следует воспользоваться для перестановки местами слов hither и yon в любой строке с любым количеством слов между ними? (При этом не стоит обращать внимание на специальные знаки пунктуации, следует только учитывать буквы верхнего и нижнего регистра и пробелы.)

Глава 7. Редактор emacs

В 1956 году сотрудник Массачусетского технологического института (MIT) Джон Маккарти (John McCarthy) разработал язык Лисп (List processing — обработка списков). В исходной концепции язык Лисп имел небольшое количество скалярных (*атомарных*) типов данных и только одну *структуру данных* — список. Списки могли содержать атомарные данные или другие списки.

Лисп поддерживал рекурсию и нечисловые данные (которыми увлекались во времена Фортрана и Кобола) и стал (по крайней мере, в Кембридже) одним из наиболее почитаемых языков реализации программного обеспечения. Среди тех, кто занимался развитием языка Лисп в MIT, были Ричард Столлмен (Richard Stallman) и Гай Стил (Guy Steele). В 1975 году они объединили свои усилия в работе над emacs, разработку которого Столлмен долгое время вел в одиночку. В данной главе рассматривается редактор emacs версии 22, реализованной Фондом свободного программного обеспечения — Free Software Foundation (GNU). Адрес главной страницы редактора emacs — www.gnu.org/software/emacs.

История

Прототипом редактора emacs послужил ряд расширенных команд или макросов, созданных для текстового редактора конца 60-х годов TECO (Text Editor and Corrector — текстовый редактор и корректор). Его происхождение отражено в имени, являющимся сокращением от Editor MACroS (макрос редактора), хотя эта аббревиатура породила и множество юмористических толкований, в числе которых ESCAPE META ALT CONTROL SHIFT (названия клавиш), Emacs Makes All Computing Simple (Emacs превращает все компьютеры в простые устройства) и недоброжелательный перевод его названия — Eight Megabytes And Constantly Swapping (постоянная подкачка страниц при восьми мегабайтах памяти).

Развитие

Со временем emacs совершенствовался, выдержав более чем 20 основных версий и дойдя до широко распространенной GNU-версии. Редактор emacs, разработанный на C, содержит полноценный интерпретатор языка Лисп и полностью поддерживает X Window System, а также работу с мышью. В своем развитии он уже ушел далеко вперед от исходных макросов TECO, но все еще является инструментом, находящимся в активной разработке. Версия 22 существенно обновлена в связи с интернационализацией: она обладает расширенным внутренним набором символов UTF-8, в четыре раза превышающим по объему Unicode, а также шрифтами и методами ввода символов с клавиатуры, позволяющими работать на более чем 30 языках. К тому же пользовательский интерфейс развивается в направлении поддержки концепции текстового процессора WYSIWYG (what you see is what you get — что видишь, то и имеешь), что облегчает новичкам освоение работы с этим редактором.

Редактор emacs всегда был существенно более развитым средством, чем простой текстовый редактор. Будучи изначально разработанным вне UNIX-среды, он не придерживается UNIX/Linux-философии. В отличие от UNIX/Linux-утилиты, которая обычно разрабатывается для выполнения какой-нибудь одной конкретной задачи и предназначена для совместного использования с другими утилитами, редактор emacs разработан в качестве средства «все в одном». Пользуясь преимуществами положенного в его основу языка программирования (Лисп), пользователи emacs предпочитают вместо применения уже существующих утилит и создания новых универсальных инструментов настраивать под свои нужды и расширять функции самого редактора. То есть они делятся друг с другом своими настроечными файлами ~/.emacs.

Задолго до появления X Window System Ричард Столлмен вложил немало сил и фантазии в создание оконной рабочей среды, используя emacs в качестве исследовательского полигона. Со временем он создал внутри emacs средства для чтения и составления сообщений электронной почты, чтения и публикации сетевых новостей, запуска команд оболочки, компилирования программ и анализирования сообщений об ошибках, запуска и отладки этих программ, а также для запуска игр. В конечном итоге появилась возможность войти в среду emacs и не выходить из нее весь день, лишь переключаясь между окнами и файлами. Если в распоряжении пользователя имеется только символьный терминал, то emacs существенно увеличивает его возможности.

При работе в среде X Window System emacs не нуждается в передаче ему управления всем дисплеем, ему обычно достаточно одного-двух окон. Но при работе с ним по-прежнему доступна и исходная символьная среда, и поэтому именно этот режим работы будет рассмотрен в данной главе.

В качестве редактора, *восприимчивого к языкам программирования*, emacs обладает специальными свойствами, которые включаются с целью оказания помощи в редактировании текста, в его форматировании с помощью утилиты nroff, в верстке с помощью системы TeX, в написании программ на Лиспе, C, Фортране и т. д. Эти наборы свойств называются *режимами*, но они не имеют ни малейшего сходства с командным режимом и режимом редактирования, использующимися в vi, vim и в других редакторах. Поскольку при работе с emacs его не нужно переключать между режимом редактирования и командным режимом, он называется редактором, *не имеющим режимов работы*.

Сравнение emacs с vim

Интересная дискуссия насчет непрекращающихся войн между редакторами изложена по адресу en.wikipedia.org/wiki/Editor_war, а сравнительные оценки emacs и vim можно найти на просторах интернета.

Как и vim, emacs является экранным редактором: он отображает экран редактируемого текста и изменяет отображаемую информацию при вводе команды или вставке нового текста. В отличие от vim emacs не требует постоянного контроля над тем, в каком режиме вы находитесь, в командном режиме или в режиме ввода: при вводе команд всегда используется клавиша CONTROL или другие специальные клавиши. В редактируемый текст emacs вставляет обычные символы (и не использует их в качестве команд), что также является характерной особенностью редактирования без

использования режимов работы. Многим людям такой подход представляется вполне удобным и естественным.

Как и vim, emacs используется для редактирования текста в рабочей области или буфере и имеет возможность записи этого буфера обратно в файл на диске по окончании работы. Но, работая с emacs, вы можете иметь множество рабочих буферов, и переключаться между ними без обязательной записи буфера на диск с последующим его считыванием с диска. Более того, можно одновременно выводить на экран несколько буферов, открывая для каждого из них отдельное окно редактора. Этот способ отображения файлов полезен при вырезании и вставке текста или когда нужно держать на видном месте все объявления в программе на C в процессе редактирования кода, который находится в другой части файла.

Как и vim, редактор emacs обладает богатым, исчерпывающим набором команд для перемещения по буферу и внесения изменений в текст. Этот набор не является застывшей твердыней, в любой момент команды можно изменить или перенастроить под свои предпочтения. Любая клавиша может быть привязана к любой команде, чтобы лучше подходить к конкретной клавиатуре или отвечать персональным запросам. Обычно все привязки клавиш к командам хранятся в пусковом файле ~/.emacs, но они могут меняться в интерактивном режиме во время сеанса редактирования. Все привязки клавиш к командам, рассматриваемые в данной главе, являются стандартными для текущих GNU-версий редактора emacs.

ИЗБЕГАЙТЕ СЛИШКОМ МАСШТАБНЫХ ИЗМЕНЕНИЙ ПРИВЯЗОК КЛАВИШ К КОМАНДАМ

Если вы измените слишком много клавишных привязок, то может получиться набор команд, который вы не сумеете запомнить. Или набор, который может не позволить вам вернуться в данном сеансе к стандартным привязкам.

И наконец, явное отличие от vim заключается в том, что emacs позволяет использовать Лисп для создания новых команд или перезаписи старых. Столлмен назвал эту особенность постоянной возможностью к расширению, но для того чтобы написать и отладить новую команду в процессе редактирования текста, нужно быть довольно смелым Лисп-программистом. Чаще всего в файл .emacs добавляются уже отлаженные команды, откуда они будут загружены в процессе запуска редактора. Опытные пользователи emacs зачастую создают режимы или рабочие окружения, которые загружаются редактором emacs при определенных условиях для решения специфических задач. Дополнительные сведения о файле .emacs даны в разделе «Файл запуска .emacs».

ЭКРАН И ОКНА EMACS

В данной главе понятие «экран» относится к экрану символьного терминала или к окну эмулятора терминала, работающему в графическом режиме. А понятие «окно» относится к окну emacs, находящемуся на этом экране.

EMACS И X WINDOW SYSTEM

Начиная с версии 19, GNU emacs располагает полноценной средой X Window System. Если emacs запускается из окна эмулятора терминала, запущенного в графической среде, то для emacs будет возвращен X-интерфейс (GUI). В данной книге графический интерфейс не рассматривается, поэтому при запуске emacs следует воспользоваться ключом –nw, чтобы попасть в текстовый интерфейс при любой среде окружения. См. далее раздел «Запуск emacs» на с. 223.

Сравнение работы редактора emacs в режиме командной строки и в графическом режиме

Данная книга посвящена работе в Linux с помощью интерфейса командной строки. А эта глава посвящена главным образом работе с emacs из интерфейса командной строки. Ряд графических функций, предлагаемых редактором emacs, рассмотрен в разделе «Графический интерфейс пользователя emacs» на с. 230.

Руководство по началу работы с emacs

Редактор emacs обладает огромным количеством свойств и множеством способов их реализации. Полное руководство насчитывает более 35 глав. Тем не менее, довольно существенный объем полезной работы можно выполнить с помощью относительно небольшого набора команд. В этом разделе рассматривается простой сеанс редактирования, объясняющий, как запустить редактор emacs и выйти из него, а также как перемещать курсор и удалять текст. Для более доходчивого изложения рассмотрение некоторых вопросов отложено или дано в сокращенном виде.

ИНТЕРАКТИВНОЕ УЧЕБНОЕ ПОСОБИЕ ПО EMACS

Редактор emacs предоставляет своим пользователям интерактивное учебное пособие. Чтобы им воспользоваться, нужно после запуска emacs нажать комбинацию CONTROL+H t. Чтобы выйти из emacs, следует нажать комбинацию CONTROL+X CONTROL+C. Если открыто более одного окна emacs, обратитесь к совету «Закрытие окна справки» на с. 239.

Запуск emacs

Для редактирования файла по имени sample с помощью emacs, используемого в качестве простого текстового редактора, введите следующую команду:

```
$ emacs -nw -q sample
```

Ключ `-nw`, который должен быть первым ключом в командной строке emacs, заставляет этот редактор отказаться от своего X-интерфейса (GUI). Ключ `-q` предписывает emacs не читать пусковой файл ~/.emacs. Отказ от чтения этого файла гарантирует, что emacs будет обладать стандартным стилем поведения и может использоваться новичками или другими пользователями, желающими обойти загрузку информации из файла .emacs.

Предыдущая команда запускает emacs, считывает файл по имени sample в буфер и отображает его содержимое на экране или в окне. Если на данный момент файла с таким именем не существует, emacs отображает пустой экран с пометкой `(New File)` в его нижней части (рис. 7.1). Если файл с таким именем существует, emacs отображает его содержимое и выдает другое сообщение (рис. 7.3). Если emacs запущен без указания имени файла в командной строке, он отображает экран приветствия, который включает информацию по использованию и список основных команд (рис. 7.2).

Изначально emacs отображает одно окно. В верхней части окна в негативном оформлении отображается строка меню, доступ к которой можно получить, используя мышь или клавиатуру. При использовании клавиатуры окно списка расширений

пунктов меню отображается с помощью клавиш F10, META+‘ (обратный апостроф) или с помощью команды META-x tmm-menubar ВВОД. Дополнительные сведения даны в разделе «Использование меню с помощью клавиатуры».

Рис. 7.1. Экран нового файла emacs

В нижней части окна emacs в негативном оформлении выводится информационная строка, называемая строкой режима (Mode Line). В минимальном формате строка режима показывает, какой именно буфер просматривается в окне, был ли этот буфер изменен, какие основные и второстепенные режимы задействованы и насколько глубоко позиционируется окно в буфере. Когда на экране появляется несколько окон, то в каждом из них отображается своя строка режима. В самой нижней части экрана emacs оставляет открытой одну строку. Это область вывода и строка мини-буфера (они сосуществуют на одной строке), которые используются для вывода сообщений и ввода специальных однострочных команд.

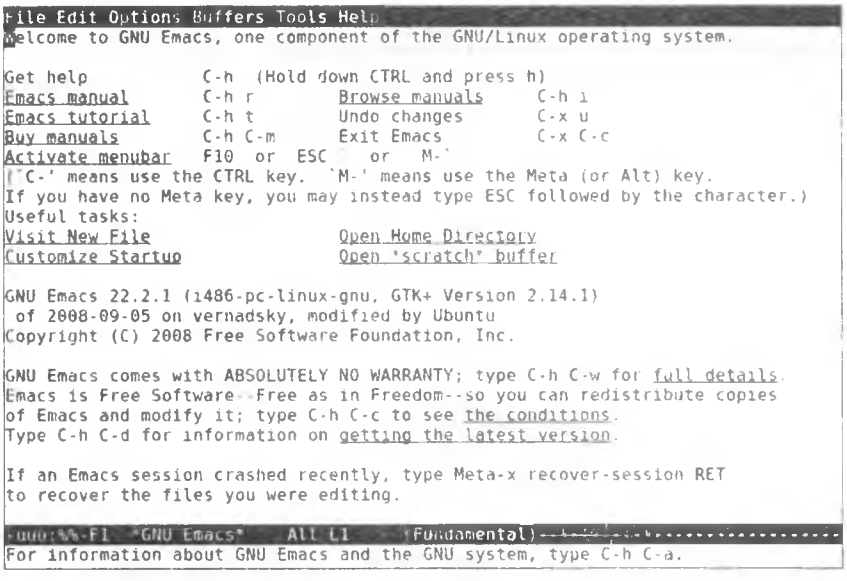

Рис. 7.2. Экран приветствия emacs

СПРАВОЧНОЕ РУКОВОДСТВО ПО EMACS

Справочное руководство по emacs доступно из самого редактора. Введите во время работы с emacs команду CONTROL+H r. После этого для прокрутки на нужный раздел используйте клавиши стрелок, а затем нажмите клавишу ВВОД. Вместо этого можно набрать m (вызвав перемещение курсора в мини-буфер), а затем набрать название раздела (пункта меню), который нужно просмотреть. Чтобы заставить emacs завершить название пункта меню, следует нажать клавишу табуляции. Автозавершение меню работает так же, как и автозавершение имени (см. раздел «Автозавершение имени»). Способы закрытия окна справки даны в совете «Закрытие окна справки» на с. 239, а дополнительные сведения об интерактивной справке даны в разделе «Интерактивная справка» на с. 239.

Например, чтобы просмотреть в интерактивном справочном руководстве раздел Minibuffer, нужно ввести команду CONTROL+H r m minibuffer ВВОД. Можно также дать команду CONTROL-H r m min TAB ВВОД.

Если при наборе команды в мини-буфер будет допущена ошибка, emacs выводит сообщение об ошибке в область вывода. Сообщение об ошибке будет выведено поверх набранной команды, но emacs через несколько секунд вернет команду на место. Краткого отображения сообщений об ошибках вполне достаточно, чтобы их прочитать перед продолжением набора команды с того места, где он был прекращен. Более подробные сведения даны на странице интерактивного справочного руководства, вызываемой через пункт меню Minibuffer (см. предыдущий совет).

Курсор находится либо в окне, либо в мини-буфере. Весь ввод и почти все редактирование происходят под курсором. При наборе обычных символов emacs вставляет их в позицию курсора. Если под курсором или справа от него находятся символы, они сдвигаются вправо по мере ввода, поэтому никакие символы не теряются.

Выход

Команда выхода из emacs вводится с помощью комбинации CONTROL+X CONTROL+C. Эту команду можно ввести практически в любой момент (в некоторых режимах сначала может потребоваться нажать комбинацию CONTROL+G). Эта команда завершит работу emacs предельно корректно, переспросив, желаете ли вы сохранить изменения, внесенные во время сеанса редактирования.

Если нужно отменить наполовину введенную команду или остановить запущенную команду до того, как она будет выполнена, следует нажать комбинацию CONTROL+G. Редактор emacs отобразит в области вывода сообщение `Quit` (Завершена) и будет ждать следующей команды.

Вставка текста

При наборе обычных (выводимых на печать) символов курсор и любые символы справа от него сдвигаются на одну позицию вправо, и новый символ вставляется в пространство, освобождаемое сдвигаемыми символами.

Удаление символов

В зависимости от используемой клавиатуры и настроек файла запуска emacs, символы могут удаляться разными клавишами и разными способами. Обычно комбинация CONTROL+D удаляет символ под курсором, то же самое делается с помощью

клавиш DELETE и DEL. Клавиша BACKSPACE обычно удаляет символ слева от курсора. Попробуйте воспользоваться каждой из этих клавиш и посмотрите, что они делают.

И ЕЩЕ ОБ УДАЛЕНИИ СИМВОЛОВ

Если инструкции, рассматриваемые в данном разделе, не работают, прочитайте дополнительную информацию по emacs, посвященный удалению символов. Для этого введите из приглашения оболочки следующую команду:

```
$ info emacs
```

Находясь в программе info, введите команду m deletion, чтобы отобразился документ, подробно описывающий порядок удаления небольших фрагментов текста. Для прокрутки документа используйте ПРОБЕЛ. Для выхода из программы info нажмите клавишу q. Ту же информацию можно получить в интерактивном справочном руководстве по emacs (по команде CONTROL-H r; см. раздел «Интерактивная справка» на с. 239).

Запустите emacs и наберите несколько строк текста. При ошибках внесите необходимые исправления, пользуясь ранее рассмотренными клавишами удаления. Клавиша ВВОД вставляет в буфер невидимый символ конца строки и возвращает курсор к левой границе на одну строку ниже. Можно вернуться за начало строки и попасть в конец предыдущей строки. На рис. 7.3 показан типовой буфер.

ПОЛЬЗУЙТЕСЬ КЛАВИШАМИ СТРЕЛОК

Иногда самым простым способом перемещения курсора является использование клавиш стрелок влево, вправо, вверх и вниз.

Перемещение курсора

Курсор можно поставить на любом символе в окне emacs и переместить окно таким образом, чтобы в нем отображалась любая часть буфера. Курсор можно перемещать вперед или назад по тексту (см. рис. 6.8 в главе 6), по различным текстовым элементам — например, по символам, по словам, по предложениям, по строкам и по абзацам. Перед любыми командами перемещения курсора можно указывать количество повторений (команда CONTROL+U, за которой следует числовой аргумент), заставляющее курсор перемещаться по тексту на указанное количество текстовых элементов. Числовые аргументы рассмотрены далее в соответствующем разделе.

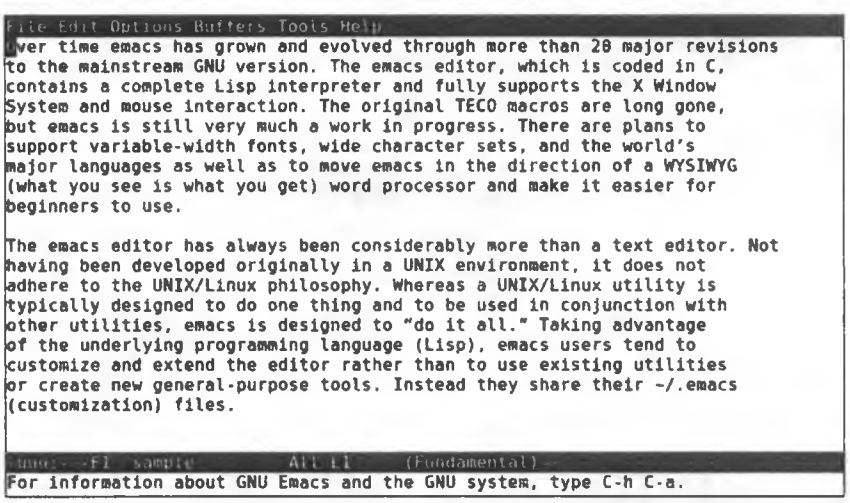

Рис. 7.3. Типовой буфер

Перемещение курсора по символам

Команда CONTROL+F

Чтобы переместить курсор вперед (вправо) на один символ, следует нажать клавишу СТРЕЛКА ВПРАВО или комбинацию клавиш CONTROL+F. Если курсор находится в конце строки, эта команда переместит его в начало следующей строки. Например, команда CONTROL+U 7 CONTROL+F переместит курсор на семь символов вперед.

Команда CONTROL+B

Чтобы переместить курсор назад (влево) на один символ, следует нажать клавишу СТРЕЛКА ВЛЕВО или комбинацию клавиш CONTROL+B. Например, команда CONTROL+U 7 CONTROL+B переместит курсор на семь символов назад. Работа команды CONTROL+B похожа на работу команды CONTROL+F (рис. 7.4).

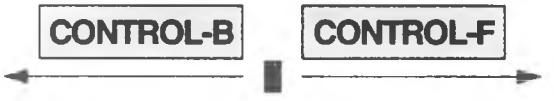

Рис. 7.4. Перемещение курсора по символам

Перемещение курсора по словам

Команда META+f

Чтобы переместить курсор на одно слово вперед, следует нажать комбинацию клавиш META+f. Для вызова этой команды нужно при нажатии клавиши f удерживать в нажатом состоянии клавишу META или ALT. Если используемая вами клавиатура не имеет ни одной из этих клавиш, нажмите клавишу ESCAPE, отпустите ее, а затем нажмите клавишу f. Эта команда оставляет курсор на первом символе, не являющемся частью того слова, с которого он начал перемещение. Команда CONTROL-U 4 META-f переместит курсор вперед на один пробел, следующий за концом четвертого слова. Дополнительные сведения даны в разделе «Клавиши: система записи обозначений и использование» на с. 231.

Команда META+b

Чтобы переместить курсор на одно слово назад, следует нажать комбинацию клавиш META+b, при этом курсор останется на первой букве того слова, с которого он начал перемещение. Если курсор был на первой букве слова, нажатие комбинации META+b приведет к перемещению курсора на первую букву предыдущего слова. Работа команды META +b похожа на работу команды META+f (рис. 7.5).

Перемещение курсора по строкам

Команды CONTROL+A, CONTROL+E, CONTROL+P и CONTROL+N

При нажатии комбинации CONTROL+A курсор перемещается в начало той строки, на которой находился, а при нажатии комбинации CONTROL+E — в конец этой строки. При нажатии клавиши СТРЕЛКА ВВЕРХ или комбинации клавиш CONTROL+P курсор перемещается на одну строку вверх, на позицию, находящуюся непосредственно над той, с которой он начал перемещение, а при нажатии клавиши СТРЕЛКА ВНИЗ или комбинации клавиш CONTROL+N курсор перемещается вниз. Как и при работе с другими клавишами перемещения курсора, вы можете для перемещения курсора сразу на несколько строк предварять нажатие комбинаций CONTROL+P и CONTROL+N нажатием комбинации CONTROL+U и вводом числового аргумента. Можно также использовать

пары этих команд для перемещения курсора вверх в начало предыдущей строки, вниз в конец следующей строки и т. д. (рис. 7.6).

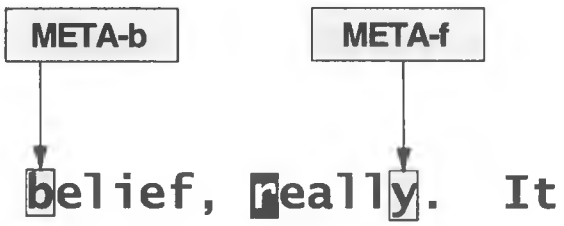

Рис. 7.5. Перемещение курсора по словам

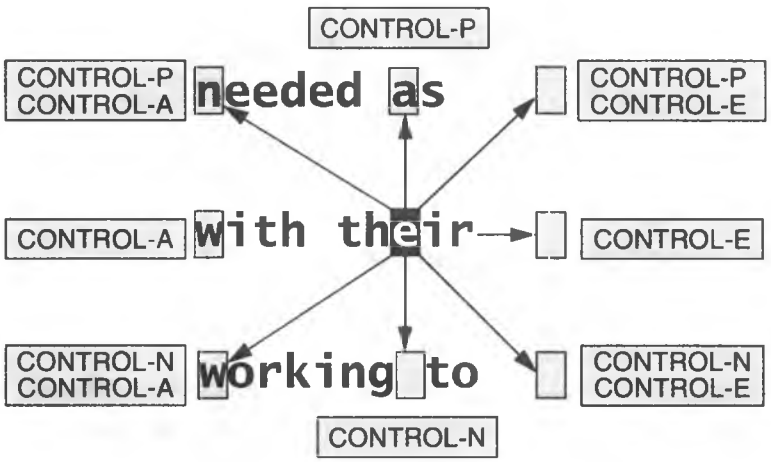

Рис. 7.6. Перемещение курсора по строкам

Перемещение курсора по предложениям, абзацам и позициям окна

Команды META+a, META+e, META+{ и META+}

При нажатии комбинации META+a курсор перемещается к началу предложения, на котором он находится, а при нажатии комбинации META+e курсор перемещается к концу этого предложения. При нажатии комбинации META+{ курсор перемещается к началу абзаца, на котором он находится, а при нажатии комбинации META+} курсор перемещается к концу этого абзаца. (Определения предложениям и абзацам даются в соответствующих, посвященных им разделах.) Перед любой из этих команд можно ставить Число повторений (команда CONTROL+U, за которой следует числовой аргумент), заставляющее курсор перемещаться на указанное количество предложений или абзацев.

Команда META+r

При нажатии комбинации META+r курсор перемещается в начало средней строки окна. Можно перед этой командой нажать комбинацию CONTROL+U и указать номер строки (здесь CONTROL+U является не признаком количества повторений, а признаком номера строки на экране). Команда CONTROL+U 0 META+r вызовет перемещение курсора в начало самой верхней (нулевой) строки окна. Команда CONTROL+U – (знак минуса) вызовет перемещение курсора в начало последней строки окна (рис. 7.7).

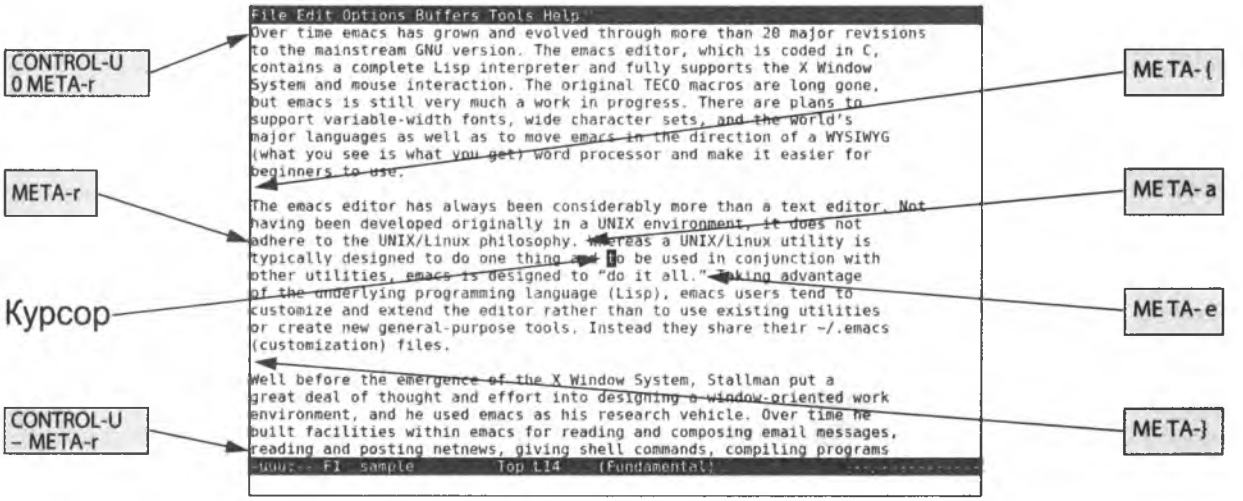

Рис. 7.7. Перемещение курсора по предложениям, абзацам и позициям окна

Редактирование в позиции курсора

Если курсор в окне выставлен на позицию, в которую нужно вводить текст, то для ввода текста не требуется никаких команд. При наборе текста emacs отображает этот текст в позиции курсора. Любой текст, находящийся ниже или правее курсора, сдвигается вправо. При вводе такого количества символов, при котором текст перейдет правую границу окна, emacs отображает символ обратного слэша (\) у правой границы окна и переносит текст на следующую строку. Символ обратного слэша появляется на экране, но не сохраняется в файле и никогда не выводится на печать. Хотя вы можете создать строки произвольной длины, некоторые инструментальные средства Linux испытывают проблемы при работе с текстовыми файлами, содержащими подобные строки. Чтобы разбить строку на две строки, нужно выставить курсор на то место, где нужно разбить строку, и нажать клавишу ВВОД.

Удаление текста

При нажатии клавиши BACKSPACE происходит удаление символа слева от курсора. При каждом нажатии клавиши BACKSPACE курсор и вся остальная часть этой строки перемещаются влево. Для объединения строки со стоящей над ней строкой нужно выставить курсор на первом символе второй строки и нажать клавишу BACKSPACE.

Для удаления символа под курсором нужно воспользоваться комбинацией CONTROL+D. При этом курсор остается на месте, но весь текст, оставшийся в строке, перемещается влево, заполняя позицию удаленного символа. Если обе эти клавишные команды не работают так, как описано здесь, обратитесь к совету «И еще об удалении символов» на с. 226.

Сохранение буфера и извлечение его содержимого

Независимо от тех изменений, которые были внесены в буфер во время сеанса работы с emacs, связанный с ним файл не будет изменен до тех пор, пока вы не сохраните содержимое буфера. Если выйти из emacs без сохранения буфера (при определенной

настойчивости emacs позволяет это сделать), файл не изменится, и emacs сбросит всю работу, проделанную в ходе сеанса.

Создание резервной копии

В процессе обратной записи в файл отредактированного содержимого буфера emacs при определенной настройке может сначала создать резервную копию исходного файла. Можно выбрать отказ от резервного копирования, одноуровневое резервное копирование (этот вариант устанавливается по умолчанию) или произвольное количество уровней резервного копирования. Имена файлов при одноуровневом резервном копировании формируются за счет добавления к исходному имени файла знака тильды (~). При многоуровневом создании резервных копий к имени файла добавляется сочетание символов .~*n*~, где *n* — это номер следующей резервной копии, начинающийся с единицы. Способ создания редактором emacs резервных копий определяется значением переменной `version-control`. Инструкции по установке значения переменной emacs изложены в разделе «Присваивание значения переменной» на с. 267.

Сохранение содержимого буфера

Текущий буфер сохраняется в связанном с ним файле с помощью комбинаций CONTROL+X CONTROL+S. Редактор emacs подтверждает успешное сохранение, выводя в область вывода соответствующее сообщение.

Посещение другого файла

Когда при редактировании файла в emacs возникает потребность в редактировании другого файла (в документации по emacs такое редактирование файла называется посещением файла — «visiting a file»), можно скопировать новый файл в новый буфер emacs путем ввода команды CONTROL+X CONTROL+F. Редактор emacs запросит имя файла, считает его в новый буфер и отобразит содержимое этого буфера в текущее окно. В одном сеансе редактирования удобнее держать открытыми два файла, чем выходить из emacs, возвращаться в оболочку, а затем запускать новую копию emacs для редактирования второго файла.

ПОСЕЩЕНИЕ ФАЙЛА С ПОМОЩЬЮ КОМАНДЫ CONTROL+X CONTROL+F

Когда для посещения файла вводится команда CONTROL+X CONTROL+F, emacs отображает имя каталога, в котором, по его предположениям, находится файл. Обычно он отображает имя рабочего каталога, но в некоторых ситуациях emacs отображает другое имя, например имя вашего личного каталога. Если это имя указывает на неправильный каталог, его нужно отредактировать. Эта команда предоставляет возможность автозавершения имени (см. раздел «Автозавершение имени» на с. 249).

Графический интерфейс пользователя emacs

Изначально редактор emacs запускался на ASCII (текстовых) терминалах с последовательными интерфейсами. Со временем, по мере появления графических дисплеев, emacs стал приобретать все большую графическую ориентированность. В версии 19 появилась полная поддержка мыши. Редактор emacs продолжает развиваться в направлении полной интернационализации, доступности и простоты использования, рассчитанной на широкий круг пользователей. Новые свойства включают в себя многоцветные линейки инструментов, смешение текста и графики, всплывающие подсказки, редактирование в режиме перетаскивания фрагментов, новые графические

интерфейсы пользователей для просмотра каталогов и выбора шрифтов. Многие из этих функций доступны только из графических интерфейсов.

ЗДЕСЬ РАССМАТРИВАЮТСЯ ВОПРОСЫ РАБОТЫ С EMACS С ИСПОЛЬЗОВАНИЕМ КЛАВИАТУРЫ

Поскольку данная книга ориентирована на использование командной строки, в этой главе рассматриваются вопросы работы с emacs с использованием клавиатуры. В данном разделе дается краткое введение в графический пользовательский интерфейс emacs, рассчитанное на тех читателей, которые хотят пойти по этому пути его использования.

Чтобы увидеть, на что похож графический режим, нужно запустить emacs без ключа -nw из окна эмулятора терминала, созданного в графической среде. Редактор emacs отобразит экран приветствия (рис. 7.8), на котором будут доступные для мыши гиперссылки на локальные файлы справок и на главную страницу сайта, посвященного проекту GNU. Графический интерфейс emacs предоставляет строку меню и линейку инструментов, чтобы можно было выбирать нужные функции с помощью мыши. Все клавиатурные команды, представленные в данной главе, работают и в том случае, если emacs отображает свой графический интерфейс.

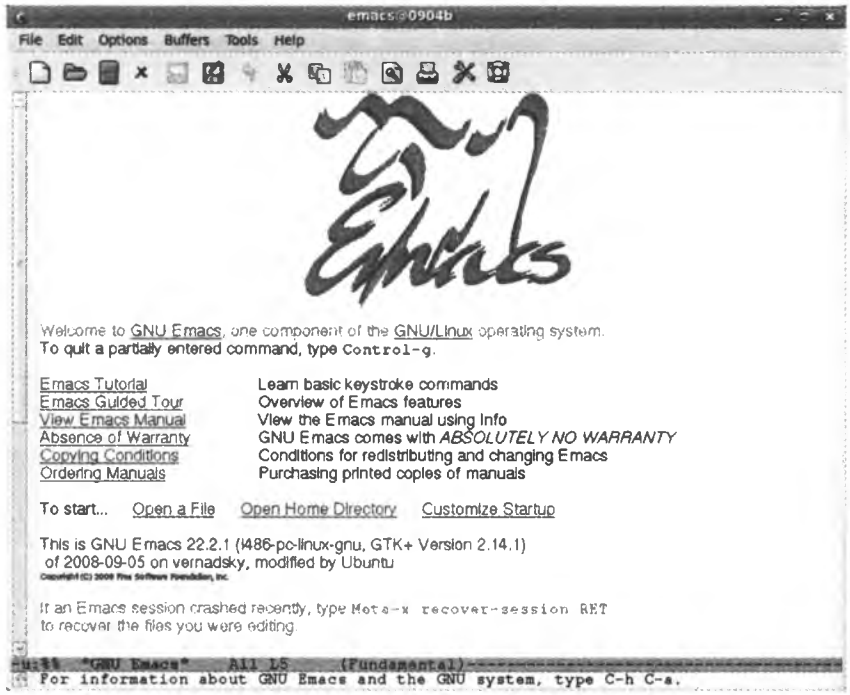

Рис. 7.8. Графический экран приветствия emacs

Основные команды редактирования

В этом разделе дается более подробный взгляд на основные команды редактирования emacs. Они предназначены для редактирования одного файла в одном окне emacs.

Клавиши: система записи обозначений и использование

Несмотря на то что emacs был приспособлен для работы на многих языках мира, в его клавиатурном вводе все еще используется и расширяется ASCII-код, при этом одно

нажатие клавиши генерирует один байт. У ASCII-клавиатур есть клавиша SHIFT, меняющая режим набора символов, и управляющая клавиша CONTROL. У некоторых клавиатур имеется также клавиша META (клавиша со значком ромба, или клавиша ALT), которая управляет восьмым битом. Для описания символа ASCII достаточно семи бит, а восьмибитный байт может быть использован для передачи дополнительной информации. Из-за обилия команд emacs, задающихся невыводимыми на печать символами, генерируемыми в регистрах CONTROL или META, Столлмен стал одним из первых разработчиков нецифровой системы записи для описания нажатий клавишных комбинаций.

Его вариант решения, который до сих пор используется в сообществе пользователей emacs, имеет вполне понятную и однозначно толкуемую форму (табл. 7.1). В нем используются заглавные буквы C и M, обозначающие соответственно удерживание клавиш CONTROL и META (или ALT), и несколько простых акронимов для наиболее распространенных специальных символов, например RET (в данной книге используется RETURN), LFD (LINEFEED — перевод строки), DEL (DELETE), ESC (ESCAPE), SPC (SPACE) и TAB. Эта система записи используется в большей части документации по emacs, включая интерактивную справку.

Таблица 7.1. Система записи обозначений клавиш в emacs

Символ	Классическая система записи emacs
(нижний регистр) a	A
(верхний регистр) SHIFT+a	A
CONTROL+a	C-a
CONTROL+A	C-a (SHIFT не используется), эквивалент CONTROL+a
META+a	M-a
META+A	M-A (SHIFT используется), отличается от M-a
CONTROL+META+a	C-M-a
META+CONTROL+a	M-C-a (используется редко)

Использование клавиш в emacs имеет ряд проблем. У многих клавиатур отсутствует клавиша META, а некоторые операционные системы сбрасывают бит META. Кроме того, набор команд emacs конфликтует с давным-давно устаревшей системой управления потоком XON-XOFF, в которой также используются комбинации CONTROL+S и CONTROL+Q.

При работе в среде операционной системы OS X многие клавиатуры не имеют клавиш META или ALT. Вопрос отсутствия клавиши META был решен путем создания дополнительной двухклавишной последовательности, начинающейся с нажатия клавиши ESCAPE, которая становится эквивалентом META. Если на используемой вами клавиатуре отсутствует клавиша META или клавиша ALT, вы можете воспользоваться двухклавишной ESCAPE-последовательностью, нажав и отпустив клавишу ESCAPE, а затем нажав клавишу, которая указывается в данной книге в комбинации с клавишей META. Например, можно набрать ESCAPE a вместо META+a или набрать ESCAPE CONTROL+A вместо CONTROL+META+a.

Столлмен считал, что система управления потоком XON-XOFF уйдет в историю и не собирался изменять набор команд emacs. Но в ответах на наиболее часто задаваемые вопросы в интерактивной справке emacs приводятся несколько путей обхода этой проблемы.

СИСТЕМА ЗАПИСИ, ИСПОЛЬЗУЕМАЯ В ДАННОЙ КНИГЕ

В данной книге после CONTROL указывается буква в верхнем регистре, а после META — буква в нижнем регистре. В любом случае вам не нужно удерживать клавишу SHIFT при вводе CONTROL- или META-символа. Хотя META-символ верхнего регистра (например, META+A) является другим символом, обычно он не настраивается под какое-либо действие или настраивается под то же самое действие, что и его аналог нижнего регистра.

Клавишные последовательности и команды

В emacs используются очень гибкие связи между клавишными последовательностями (одной или несколькими клавишами, нажимаемыми вместе или последовательно для вызова команд emacs) и командами, что дает широкие возможности для настройки их в соответствии с вашими личными предпочтениями. Вы можете переводить к другим командам или заново привязывать клавишную последовательность и заменять или перепрограммировать команды.

Хотя во многих документах, касающихся emacs, опускаются подробности и о клавишных последовательностях говорится как о командах, важно понимать, что базовые механизмы обработки отделены от клавишных последовательностей и что вы можете изменять поведение клавишных последовательностей и команд. Дополнительные сведения даны в разделе «Настройка emacs» на с. 266.

META+x: запуск команды без ее привязки к клавише

Привязка клавиш (то есть таблицы или вектора, используемые emacs для перевода клавишной последовательности в команду; см. раздел «Изменение привязки клавиш» на с. 268), уже перенасыщена, и привязать каждую команду к клавишной последовательности зачастую не представляется возможным. Каждую команду можно выполнить путем ввода ее имени, перед которым следует нажать комбинацию META+x. При нажатии META+x редактор emacs приглашает ввести команду в область вывода. После ввода имени команды и нажатия клавиши ВВОД редактор выполняет команду.

Автозавершение

Когда у команды отсутствует общепринятая клавишная последовательность, то для нее иногда дается описание в виде META+x *имя_команды*. Редактор emacs для многих запрашиваемых им ответов предлагает автозавершение. После ввода части ответа на приглашение нажмите клавишу ПРОБЕЛ или клавишу TAB, чтобы заставить emacs провести автозавершение соответственно текущего слова или всей команды. Вызов автозавершения, когда уже введено достаточно символов для однозначного толкования команды, или ввод вопросительного знака (?) приводят к открытию списка завершения (Completion List), в котором отображается список команд, из которых можно сделать выбор. В данном случае автозавершение работает точно так же, как и автозавершение имен (см. раздел «Автозавершение имени» на с. 249).

Числовые аргументы

В emacs некоторые из команд редактирования воспринимают в качестве Числа повторений числовые аргументы. Эти аргументы следует вводить непосредственно перед клавишной последовательностью, предназначенной для вызова команды. Отсутствие аргумента почти всегда означает Число повторений равное единице. Даже обычный

символ алфавита может иметь числовой аргумент, что означает «Ввести этот символ несколько раз». Чтобы дать команде числовой аргумент, воспользуйтесь любой из следующих технологий:

- Нажмите клавишу META с каждой цифрой из диапазона 0–9 или со знаком минус (–). Например, чтобы вставить 10 символов z, наберите META+1 META+0 z.
- Воспользуйтесь комбинацией CONTROL+U для начала строки цифр, включая знак минус. Например, чтобы переместить курсор вперед на 20 слов, наберите CONTROL+U 20 META+f.

Команда CONTROL+U

Для удобства пользования применение команды CONTROL+U, когда за ней не следует одна или несколько цифр, приводит к умножению на 4. Например, ввод команды CONTROL+U r означает вставку `rrrr` (4 * 1), а ввод команды CONTROL+U CONTROL+U r означает вставку `rrrrrrrrrrrrrrrr` (4 * 4 * 1). Для быстрой частичной прокрутки высокого окна может пригодиться многократное использование последовательности команд CONTROL+U CONTROL+V для прокрутки вниз на четыре строки, команд CONTROL+U META+v для прокрутки вверх на четыре строки, CONTROL+U CONTROL+U CONTROL+V для прокрутки вниз на 16 строк или команд CONTROL+U CONTROL+U META+v для прокрутки вверх на 16 строк.

Указатель и курсор

Указатель — это место в буфере, где происходит редактирование и где позиционируется курсор. Строго говоря, указатель соответствует левому краю курсора и находится *между* двумя символами. У каждого окна есть свой собственный указатель, но курсор у редактора только один. Когда курсор находится в окне, перемещение курсора приводит также и к перемещению указателя. Переключение курсора на другое место за пределами окна не приводит к изменению места указателя в этом окне, и когда курсор будет переключен обратно на это окно, он окажется в том самом месте, где находится указатель.

Все ранее рассмотренные команды перемещения курсора перемещают также и указатель.

Прокрутка буфера

Команды CONTROL+V, META+v и CONTROL+L

В большинстве случаев размер буфера значительно превышает размер окна, через которое просматривается его содержимое, поэтому нужен способ перемещения отображаемой части вверх-вниз, чтобы интересующая часть попала в окно. Прокрутка вперед означает перемещение текста вверх с появлением новых строк в нижней части окна. Для прокрутки вперед на одно окно (за минусом двух строк для связки содержимого) следует воспользоваться комбинацией CONTROL+V или клавишей PAGE DOWN. Прокрутка назад означает перемещение текста вниз, с появлением новых строк в верхней части окна. Для прокрутки назад на одно окно (опять за минусом двух строк для связки содержимого) следует воспользоваться комбинацией META+v или клавишей PAGE UP. Нажатие клавиши CONTROL+L приводит к очистке и перерисовке экрана и к установке курсора на среднюю строку окна. Эта команда может пригодиться при возникновении помех на экране.

Числовые аргументы для команды CONTROL+V или команды META+v имеют смысл «прокрутить на указанное количество строк», например команда CONTROL+U 10 CONTROL+V означает прокрутить вперед на десять строк. Числовой аргумент для команды CONTROL+L означает «прокрутить текст, чтобы курсор оказался на данной строке окна», где 0 означает самую верхнюю строку, а –1 означает самую нижнюю строку, которая находится прямо над Строкой режима. Если при вводе команды CONTROL+P или команды CONTROL+N вы перешли за границы отображаемого в тексте окна, прокрутка осуществляется автоматически.

Команды META+< и META+>

Используя команду META+<, можно переместить курсор в начало буфера, а с помощью команды META+> можно переместить курсор в конец буфера.

Удаление текста

Разница между удалением и стиранием

При стирании текста от него можно избавиться или переместить в область хранения и по желанию чуть позже вернуть его назад. Термин *удаление* (delete) означает *полное избавление*, а термин *стирание* (kill) означает *перемещение в область хранения*. Область хранения, которая называется *кольцевым буфером стирания*, может содержать несколько фрагментов стертого текста. Текст, находящийся в кольцевом буфере стирания, можно использовать несколькими способами (см. раздел «Вырезка и вставка: восстановление стертого текста» на с. 244).

Команды META+d и CONTROL+K

Команда META+d стирает символы от курсора и до конца текущего слова. Команда CONTROL+K стирает символы от курсора и до конца текущей строки. Она не удаляет завершающий строку символ перевода строки, оставляя указатель и курсор по левую сторону этого символа. Такая настройка редактора позволяет переместить курсор в самую левую позицию строки с помощью команды CONTROL+A, стереть всю строку с помощью команды CONTROL+K, а затем тут же приступить к набору строки взамен стертой, обходясь без открытия пустого пространства для новой строки. Еще одним следствием этого является то, что для стирания строки и ликвидации пустого пространства нужно переместить курсор в начало строки и ввести команду CONTROL+K CONTROL+K (или команду CONTROL+U 2 CONTROL+K).

Поиск текста

Редактор emacs позволяет вести поиск текста следующими способами:

- Поиск символьной строки с последовательным уточнением.
- Поиск соответствия регулярному выражению с последовательным уточнением (возможен, но применяется редко).
- Поиск целиком всей символьной строки.
- Поиск соответствия всему регулярному выражению.

Все четыре вида поиска в буфере можно запускать как вперед, так и назад. Поиск всей строки или соответствия всему регулярному выражению работает точно так же,

как и в других редакторах. Поиск начинается только после полного ввода строки поиска. В отличие от этого поиск с последовательным уточнением начинается с набором первого символа строки поиска и продолжает работать при наборе следующих символов. По началу такой подход к поиску может показаться странным, но, как ни удивительно, он весьма удобен.

Поисковые операции с последовательным уточнением

Команды CONTROL+S и CONTROL+R

Выбор направления и запуск поиска осуществляются одной командой. Команда CONTROL+S запускает поиск с последовательным уточнением вперед, а команда CONTROL+R запускает такой же поиск в обратном направлении.

При запуске поиска с последовательным уточнением emacs выводит в область вывода приглашение на ввод `I-search:`. После ввода символа редактор тут же ведет поиск этого символа в буфере. Если символ найден, emacs перемещает на него указатель и курсор, чтобы показать вам действие поиска. Если поиск будет неудачен, emacs сообщит вам об этом.

После ввода в строку поиска каждого последующего символа вы можете предпринять одно из нескольких действий, в зависимости от результата поиска, полученного на данный момент. Далее рассматриваются результаты поиска и соответствующие им действия.

- В результате поиска найдена нужная вам строка, а курсор установлен справа от нее. Можно остановить поиск и оставить курсор на его новой позиции, нажав клавишу ВВОД. (Любая, не связанная с поиском команда emacs, также приведет к остановке поиска, но вспомнить о такой команде может оказаться нелегко. Новичку проще нажать ВВОД.)
- Строка, найденная в результате поиска, вам не нужна. Вы можете уточнить строку поиска, добавив к ней еще одну букву. Чтобы найти в буфере новый экземпляр искомой строки, можно нажать комбинацию CONTROL+R или CONTROL+S, а для остановки поиска и отмены дальнейшего перемещения курсора можно нажать ВВОД.
- В результате поиска был пройден весь буфер до его начала или конца, и редактор выдал сообщение о неудавшемся поиске — «Failing I-Search». Можно продолжить работу, выбрав один из следующих путей:
  - Если в строке поиска допущена опечатка, нажать нужное количество раз клавишу BACKSPACE, чтобы удалить символы из строки поиска. Текст и курсор в окне будут постепенно возвращаться назад по мере удаления символов.
  - Если нужно перейти через начало или конец буфера и продолжить поиск, можно заставить поиск преодолеть эту границу, нажав комбинацию CONTROL+R или CONTROL+S.
  - Если в результате поиска искомая строка так и не была найдена, но вы желаете оставить курсор на его текущей позиции, нужно нажать ВВОД для остановки поиска.
  - Если поиск пошел не по намеченному плану и нужно просто вернуться назад к тому месту, с которого он начался, следует нажать комбинацию CONTROL+G, генерирующую код завершения. При неудавшемся поиске однократное нажатие комбинации CONTROL+G возвращает все символы, которые не могли

быть найдены, в строку поиска. Если это действие вернуло вас к тому месту, с которого вы хотите продолжить поиск, вы снова можете добавить символы в строку поиска. Если вы не хотите продолжить поиск с этой позиции, следует нажать CONTROL+G еще раз для остановки поиска и возвращения курсора в его исходную позицию.

Поиск без последовательного уточнения

Команды CONTROL+S ВВОД и CONTROL+R ВВОД

Если вы предпочитаете провести удачный или неудачный поиск без показа всех промежуточных результатов, можно дать команду на поиск без последовательного уточнения, воспользовавшись комбинацией CONTROL+S ВВОД для поиска в прямом или комбинацией CONTROL+R ВВОД для поиска в обратном направлении. Поиск не начнется, пока в ответ на приглашение emacs не будет введена строка поиска и еще раз не будет нажата клавиша ВВОД. Ни одна из этих команд не осуществляет переход через границу буфера к его началу или концу.

Поисковые операции с использованием регулярных выражений

Используя регулярные выражения, в emacs можно проводить поиск как с последовательным уточнением, так и без него. Для запуска поиска с использованием регулярного выражения используются команды, показанные в табл. 7.2.

Таблица 7.2. Поиск соответствий регулярному выражению

Команда	Результат
META+CONTROL+s	Поиск соответствия регулярному выражению с последовательным уточнением в прямом направлении с приглашением на ввод одиночных символов
META+CONTROL+r	Поиск соответствия регулярному выражению с последовательным уточнением в обратном направлении с приглашением на ввод одиночных символов
META+CONTROL+s ВВОД	Приглашение на ввод всего регулярного выражения с последующим поиском в прямом направлении
META+CONTROL+r ВВОД	Приглашение на ввод всего регулярного выражения с последующим поиском в обратном направлении

Использование меню с помощью клавиатуры

В этом разделе рассматривается использование клавиатуры для выбора команд из меню emacs (см. рис. 7.1). В графической среде для этих целей можно также воспользоваться мышью. Использование меню соответствует нахождению emacs в Основном режиме (см. раздел «Основные режимы: редактирование с восприимчивостью к языкам программирования» на с. 256). Например, при редактировании программы на языке C строка меню включает меню этого языка, в котором содержатся команды, имеющие отношение к редактированию и отступам кода программы на языке C.

Чтобы выбрать команду из меню, сначала нужно нажать функциональную клавишу F10, комбинацию META+‘ (Обратный апостроф) или META+x `tmm-menubar` ВВОД. Редактор emacs отобразит список расширений пунктов меню, заполненный пунктами

меню верхнего уровня — File (Файл), Edit (Редактировать), Options (Настройки) и т. д., с текущим выбранным пунктом, отображенным в мини-буфере. На рис. 7.9 показано окно списка расширений пунктов меню, где в качестве текущего выбранного пункта в мини-буфере показан пункт File (Файл).

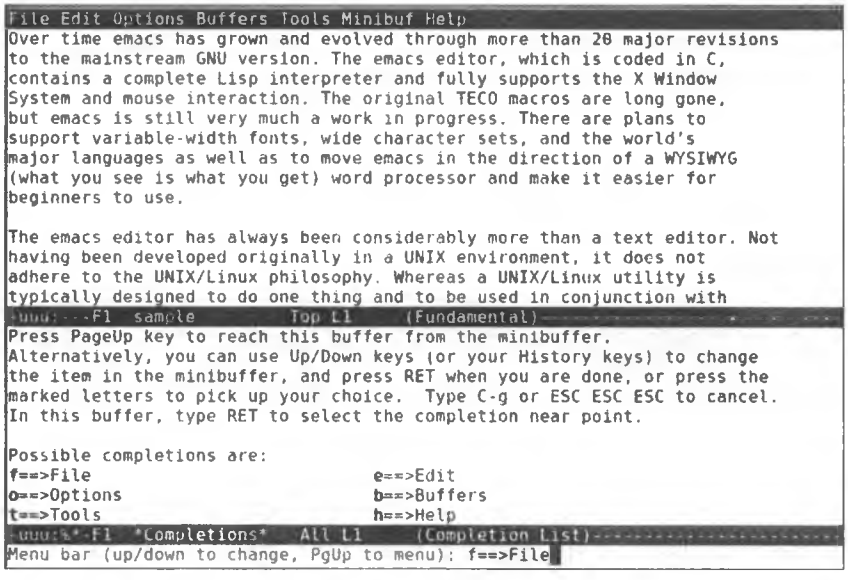

Рис. 7.9. Окно Списка расширений пунктов меню верхнего уровня

Когда открыто окно списка расширений пунктов меню, можно предпринять одно из следующих действий:

- Прекратить работу по выбору пунктов меню, нажав CONTROL+G или ESCAPE ESCAPE ESCAPE. Дисплей вернется к тому состоянию, в котором он был до открытия окна списка расширений пунктов меню.
- Использовать клавиши СТРЕЛКА ВВЕРХ и СТРЕЛКА ВНИЗ для отображения следующих пунктов меню в мини-буфере. Нажать клавишу ВВОД для выбора отображаемого пункта меню.
- Набрать однобуквенное сокращение выбранного пункта, показанное в окне списка расширений пунктов меню, чтобы выбрать нужный пункт. Клавишу ВВОД нажимать не нужно.
- Нажать клавишу PAGE UP или комбинацию клавиш META+v для перемещения курсора в окно списка расширений пунктов меню. Воспользоваться клавишами стрелок для перемещения между пунктами. Нажать ВВОД для выбора пункта, на котором находится курсор. Для возвращения из этого окна назад в мини-буфер можно нажать ESCAPE ESCAPE ESCAPE.

После выбора пункта из меню верхнего уровня emacs отображает соответствующее меню второго уровня в окне списка расширений пунктов меню. Для выбора из этого меню следует повторить одно из предшествующих действий. Когда будет сделан завершающий выбор, emacs закрывает окно списка расширений пунктов меню и предпринимает выбранное действие. Дополнительные сведения даны в интерактивном справочном руководстве и доступны при выборе пункта меню Menu Bar (см. совет «Справочное руководство по emacs» на с. 225).

Интерактивная справка

Команда CONTROL+H

Справочная система emacs доступна в любой момент. При использовании привязки клавиш по умолчанию ее можно запустить с помощью комбинации CONTROL+H. После этого справочная система выдаст приглашение на ввод однобуквенной команды справки. Если вы не знаете нужной команды справки, наберите знак вопроса (?) или нажмите комбинацию CONTROL+H, чтобы переключиться с текущего окна на список команд справки с однострочным описанием, после чего emacs опять запросит ввод однобуквенной команды справки. Если вы решите, что справка вам все же не нужна, нажмите комбинацию CONTROL+G для прекращения запроса справки и возвращения к прежнему буферу.

Если выводимая справка занимает всего одну строку, она появляется в области вывода. Если справка состоит из нескольких строк, она появляется в своем собственном окне. Для прокрутки буфера вперед и назад следует воспользоваться комбинациями CONTROL+V и META+v (см. раздел «Прокрутка буфера» на с. 234). Между окнами курсор можно перемещать с помощью команды CONTROL+X o (буква «o» в нижнем регистре). Работа с несколькими окнами рассмотрена в разделе «Окна» на с. 253.

ЗАКРЫТИЕ ОКНА СПРАВКИ

Чтобы удалить окно справки в то время, когда курсор находится в окне, содержащем редактируемый текст, следует ввести команду CONTROL+X 1 (единица). Можно также переместить курсор в окно справки (CONTROL+X o [буква «o» в нижнем регистре]) и ввести команду CONTROL+X 0 (нуль) для удаления текущего окна.

Если справка отображается в окне, занимающем весь экран, что бывает при вводе команды CONTROL+H n (новости emacs) или CONTROL+H t (учебное пособие по emacs), можно стереть буфер справки вводом команды CONTROL+X k или переключением буферов путем ввода команды CONTROL+X b (обе они рассматриваются в разделе «Буферы» на с. 252).

На многих терминалах клавиша BACKSPACE или клавиша СТРЕЛКА ВЛЕВО генерирует такой же код, что и комбинация CONTROL+H. Если вы забудете, что работаете с emacs, и попытаетесь вернуться на несколько символов назад, то можете непреднамеренно войти в систему справки. Это действие не представляет опасности для редактируемого буфера, но может сбить с толку исчезновением содержимого окна и отсутствием четкого представления о том, как все восстановить. Когда появляется приглашение команды справки, можно ввести команду CONTROL+G для удаления приглашения и возвращения к редактируемому буферу. Некоторые пользователи выбирают привязку вызова справки к другой клавише (см. раздел «Изменение привязки клавиш» на с. 268). Список некоторых команд справки приведен в табл. 7.3.

Таблица 7.3. Команды справки

Команда	Вид предоставляемой справки
CONTROL+H a	Приглашение на ввод строки и вывод списка команд, в чьем имени есть эта строка
CONTROL+H b	Вывод длинной таблицы всех действующих на данный момент клавиш, привязанных к командам

продолжение ⇩

Таблица 7.3 (*продолжение*)

Команда	Вид предоставляемой справки
CONTROL+H c *клавишная_последовательность*	Вывод имени команды, привязанной к *клавишной_последовательности*. Допускается несколько клавишных последовательностей. Для длинной клавишной последовательности, в которой распознается только первая часть, команда описывает только первую часть и без оповещения вставляет нераспознанную часть в буфер. Это может произойти с трехсимвольными функциональными клавишами (F1, F2 и другими, имеющимися на клавиатуре), генерирующими последовательности символов типа ESCAPE [SHIFT
CONTROL+H f	Приглашение на ввод имени Лисп-функции и вывод документации по этой функции. Поскольку команды являются Лисп-функциями, с этой командой можно использовать и имя команды
CONTROL+H i	Вывод главного меню утилиты info (см. раздел « info: Отображение информации об утилитах» на с. 56), позволяющего просмотреть документацию по emacs и другим инструментальным средствам
CONTROL+H k *клавишная_последовательность*	Вывод имени команды, привязанной к *клавишной_последовательности* и соответствующей документации. (См. описание команды CONTROL+H c)
CONTROL+H l (буква «l» в нижнем регистре)	Вывод последних 100 набранных символов. Запись сохраняется после первоначального преобразования сигнала с клавиатуры в символы. Если таблица преобразования сигнала с клавиатуры перенастроена под ваши потребности, нужно будет мысленно провести обратное преобразование
CONTROL+H m	Вывод документации и специальных клавишных привязок для текущего Основного режима (Text, C, Fundamental и т. д.; см. раздел «Выбор основного режима» на с. 257)
CONTROL+H n	Вывод файла новостей emacs, в котором перечисляются последние изменения, внесенные в emacs, отсортированные по степени их свежести
CONTROL+H r	Вывод справочного руководства по emacs
CONTROL+H t	Запуск сеанса работы с учебным пособием по emacs
CONTROL+H v	Приглашение на ввод имени Лисп-переменной и вывод документации по этой переменной
CONTROL+H w	Приглашение на ввод имени команды и идентификация любой привязанной к ней клавишной последовательности. Допускается использование нескольких клавишных последовательностей. (См. описание команды CONTROL+H c)

По мере усвоения этого краткого описания, вы сможете использовать систему справки для просмотра внутренней Лисп-системы emacs. Для тех, кому это интересно, следующий список строковых значений, составленный Столлменом, соответствует многим именам, встречающимся в Лисп-системе. Чтобы посмотреть на внутреннюю функциональность emacs, любое из этих строковых значений можно использовать с командой CONTROL+H a (вывод списка команд справочной системы) или с командой META+x apropos (вывод приглашения на ввод строки и вывод список переменных, чье имя содержит эту строку).

Backward	Dir	insert	previous	View
Beginning	Down	Kill	region	What
Buffer	End	Line	register	Window
Case	File	List	screen	Word

Change	Fill	mark	search	Yank
Char	Find	mode	sentence	
Defun	Forward	next	set	
Delete	Goto	page	sexp	
Describe	Indent	paragraph	up	

Расширенное редактирование

Основные команды emacs подходят для решения многих задач редактирования, но серьезные пользователи быстро почувствуют потребность в дополнительных возможностях. В этом разделе представлены некоторые расширенные возможности emacs.

Отмена изменений

Сеанс редактирования начинается с чтения файла в буфер emacs. После чтения содержимое буфера полностью соответствует содержимому файла. По мере вставки текста и ввода команд редактирования, содержимое буфера все больше отличается от содержимого файла. Если вы удовлетворены внесенными коррективами, измененный буфер можно записать обратно в файл и завершить сеанс.

Ближе к левому краю Строки режима (см. рис. 7.1) расположен индикатор, показывающий состояние изменения отображаемого в окне буфера. Этот индикатор может отображать три состояния: -- (не изменен), ** (изменен) и %% (режим только для чтения).

Редактор emacs сохраняет запись всех нажатых вами клавиш (текстовых и командных) с самого начала сеанса редактирования и вплоть до определенного предела (по умолчанию — 20 000 символов). Если этот предел не превышен, то для данного буфера возможен откат в пределах всего сеанса, по одному изменению за одну операцию отката. Если используется несколько буферов (см. раздел «Буферы» на с. 252), то у каждого буфера есть своя собственная запись отката.

Команде отмены придается такая важность, что для нее имеется резервная клавиатурная последовательность, на тот случай, если клавиатура не сможет без осложнений обработать первую последовательность. В качестве этих двух последовательностей используются CONTROL+_ (знак подчеркивания, эта последовательность на старых клавиатурах ASR-33 TTY служила в качестве клавиши СТРЕЛКА ВЛЕВО) и CONTROL+X u.

При использовании CONTROL+_ редактор emacs отменяет последнюю команду и перемещает курсор на позицию изменений в буфере, чтобы было видно, что именно произошло. Если нажать CONTROL+_ второй раз, будет отменена следующая по обратному отсчету команда и т. д. Если продолжить использование CONTROL+_, то со временем буфер вернется к своему исходному, не измененному состоянию, и индикатор в Строке режима, показывавший **, изменится на --.

Если череду команд отмены разбить вводом текста или вводом любой другой команды кроме отмены, то все обратные изменения, внесенные чередой команд отмены, становятся частью записи изменений и могут быть сами отменены. Такая стратегия предлагает способ возврата нескольких или всех отмененных операций. Если станет

понятно, что отмены зашли слишком далеко, нужно ввести команду (из разряда безвредных, не вносящих изменений в буфер, например CONTROL+F) и приступить к отмене ваших изменений в обратном направлении. В табл. 7.4 приведены некоторые примеры использования команд отмены.

Таблица 7.4. Команды отмены

Команда	Результат
CONTROL+_	Отмена последнего изменения
CONTROL+_ CONTROL+F CONTROL+_	Отмена последнего изменения с его последующим возвратом
CONTROL+_ CONTROL+_	Отмена двух последних изменений
CONTROL+_ CONTROL+_ CONTROL+F CONTROL+_ CONTROL+_	Отмена двух последних изменений с их последующим возвратом
CONTROL+_ CONTROL+_ CONTROL+F CONTROL+_	Отмена двух изменений с последующим возвратом самого последнего отмененного изменения

Если вы не помните, каким было последнее внесенное изменение, то можно ввести команду CONTROL+_ и отменить его. Если выяснится, что это изменение нужно внести, наберите CONTROL+F CONTROL+_, чтобы снова применить это изменение. Если вы случайно внесли в буфер ненужное изменение, то можете вводить команду CONTROL+_ до тех пор, пока индикатор в Строке режима снова не покажет символы --.

Если буфер полностью разрушен и нужно начать все с нуля, введите команду META+x `revert-buffer`, чтобы отказаться от текущего содержимого буфера и заново считать в него связанный с ним файл. При этом редактор emacs потребует подтвердить ваши намерения.

Указатель, метка и область

Указатель — это текущая позиция редактирования в буфере. Указатель можно перемещать по буферу путем перемещения курсора. В буфере можно также установить маркер, называемый *меткой*. Непрерывная последовательность символов между указателем и меткой (любая из них может быть первой) называется *областью*. Многие команды распространяют свое действие на область буфера, а не просто на символы около указателя.

Перемещение метки и создание области

Команды CONTROL+@, CONTROL+SPACE и CONTROL+X CONTROL+X

Метку переметить сложнее, чем указатель. После установки метка может быть перемещена только путем установки ее в каком-нибудь другом месте. У каждого буфера есть только одна метка. Команда CONTROL+@ (или CONTROL+SPACE) устанавливает метку явным образом в то место, где в данный момент находится курсор (и указатель). Некоторые клавиатуры генерируют команду CONTROL+@ при наборе CONTROL+Q. Хотя эта комбинация и не является резервной привязкой к команде, бывает, что она применяется в качестве удобной альтернативы. Для смены местами указателя и метки (и перемещения курсора на новый указатель) можно воспользоваться командой CONTROL+X CONTROL+X.

Для создания области курсор (и указатель) обычно перемещают на один край нужной области, с помощью команды CONTROL+@ устанавливают метку, а затем перемещают курсор (и указатель) к другому краю области. Если вы забыли, где поставлена метка, курсор можно вернуть на нее повторным вводом команды CONTROL+X CONTROL+X. Курсор можно перемещать вперед и назад, многократно вводя команды CONTROL+X CONTROL+X, чтобы более ясно обозначить для себя выбранную область.

Если вас не устраивает граница области, можно с помощью команды CONTROL+X CONTROL+X поменять местами с одного конца области на другой указатель и метку, а затем переместить указатель. Можно продолжить эти действия, пока область не будет удовлетворять вашим запросам.

Операции с областью

В табл. 7.5 приводится подборка команд, работающих с областью. Полный список этих команд выводится по команде CONTROL+H a region.

Таблица 7.5. Операции с областью

Команда	Результат
META+w	Копирование области в кольцевой буфер стирания без ее удаления
CONTROL+W	Стирание области
META+x `print-region`	Отправка области на принтер
META+x `append-to-buffer`	Приглашение на ввод имени буфера и добавление области к этому буферу
META+x `append-to-file`	Приглашение на ввод имени файла и добавление области к этому файлу
META+x `capitalize-region`	Перевод букв области в верхний регистр
CONTROL+X CONTROL+L	Перевод букв области в нижний регистр

Кольцевой буфер метки

При каждой установке метки в буфере можно также поместить прежнее место установки метки в *кольцевой буфер метки*. Он организован как список FIFO (первым пришел — первым ушел), сохраняющий 16 самых последних точек установки метки. У каждого буфера есть свой кольцевой буфер метки. Польза от записи недавней истории метки заключается в том, что зачастую требуется быстро перейти на прежнее место установки метки. Переход на место, указанное в кольцевом буфере метки, может быть более быстрым и простым, чем прокрутка или поиск по всему буферу того места, где вносились изменения.

Команда CONTROL+U CONTROL+@

Чтобы проделать обратный путь по бывшим местам установки метки, нужно (один или несколько раз) воспользоваться командой CONTROL+U CONTROL+@. При каждом вводе команды emacs осуществляет следующие действия:

- Перемещает указатель (и курсор) на текущее место установки метки.
- Сохраняет текущую позицию метки в *самом дальнем* конце кольцевого буфера метки.

- Забирает из кольцевого буфера самую свежую (самую недавнюю) запись и устанавливает метку.

Каждая дополнительная команда CONTROL+U CONTROL+@ заставляет emacs перемещать указатель и курсор на место, которое указывается в предыдущей записи кольцевого буфера метки.

Хотя этот процесс может показаться сложным, на самом деле он просто обеспечивает безопасное перемещение на предыдущее место метки. Безопасность основана на том, что перед каждым началом перемещения точка старта циркулирует по кольцевому буферу, где до нее нетрудно добраться еще раз. За счет повторного ввода команды CONTROL+U CONTROL+@ можно перейти на все предыдущие места установки метки, сохраненные в кольцевом буфере. Можно пройти это кольцо произвольное количество раз, остановившись в нужном месте.

Автоматическая установка метки

Некоторые команды устанавливают метку автоматически. Идея заключается в том, чтобы оставить закладку перед перемещением указателя на большие расстояния. Например, команда META+> устанавливает метку перед переходом в конец буфера. После этого вы можете вернуться в стартовую позицию с помощью команды CONTROL+U CONTROL+@. Точно так же работают команды поиска. Чтобы избежать сюрпризов, как при явной, так и при неявной установке, в области вывода появляется сообщение об установке метки: `Mark Set`.

Вырезка и вставка: восстановление стертого текста

Напомним, что стертый текст не исчезает, а помещается в кольцевой буфер стирания. В этом буфере хранятся последние 30 фрагментов стертого текста, и он виден из всех буферов. Извлечение текста из кольцевого буфера стирания называется его *восстановлением* (yanking). Значение этого термина в emacs имеет противоположный смысл тому, который использовался в vim: там под термином *yanking* подразумевалась *выборка* текста из буфера и *помещение* выбранного текста в буфер с помощью команды Поместить. Стирание и восстановление являются примерным аналогом вырезки и вставки, служа главным механизмом emacs для перемещения и копирования текста. В табл. 7.6 приводятся наиболее востребованные команды стирания и восстановления.

Таблица 7.6. Наиболее востребованные команды стирания и восстановления

Команда	Результат
META+d	Стирание символов до конца текущего слова
META+D	Стирание символов от начала предыдущего слова
CONTROL+K	Стирание символов до конца строки, за исключением символа перевода строки
CONTROL+U 1 CONTROL+K	Стирание символов до конца строки, включая символ перевода строки
CONTROL+U 0 CONTROL+K	Стирание символов от начала строки
META+w	Копирование области в кольцевой буфер стирания без ее удаления из буфера

Команда	Результат
CONTROL+W	Стирание области
META+z *char*	Стирание до следующего появления слова *char*
CONTROL+Y	Восстановление самого последнего стертого текста в текущем буфере на месте указателя; установка метки на начало этого текста, а указателя и курсора — на его конец; для смены местами указателя и метки нужно следом ввести команду CONTROL+Y
META+y	Удаление только что восстановленного текста, прокрутка кольцевого буфера стирания и восстановление его следующей записи (только после CONTROL+Y или META+y)

Чтобы переместить две строки текста, нужно переместить указатель на начало первой строки, а затем ввести команду CONTROL+U 2 CONTROL+K, чтобы стереть две строки. Затем нужно переместить указатель на целевую позицию и ввести команду CONTROL+Y.

Для копирования двух строк текста нужно переместить указатель на начало первой строки и ввести команды CONTROL+U 2 CONTROL+K CONTROL+Y, чтобы стереть строки, а затем тут же их восстановить. Затем нужно переместить указатель на целевую позицию и ввести команду CONTROL+Y.

Для копирования большого фрагмента буфера нужно установить область, покрывающую этот фрагмент, и ввести команду CONTROL+W CONTROL+Y, чтобы стереть область и восстановить ее на прежнем месте. Затем нужно переместить указатель на целевую позицию и ввести команду CONTROL+Y. Можно также установить область и воспользоваться командой META+w для копирования области в Кольцевой буфер стирания.

Кольцевой буфер стирания — FIFO-список фиксированной длины, в котором каждая новая запись при заполненном списке заставляет избавляться от самой старой записи. Простые операции вырезки и вставки проводятся, как правило, только с самыми свежими записями. Более старые записи хранятся для того, чтобы дать вам время изменить свое мнение о необходимости их удаления. Если вы передумаете, то можно будет «покопаться» в кольцевом буфере стирания, как на археологических раскопках, переместиться назад в прошлое и докопаться до слоя стертого материала, чтобы скопировать определенную запись назад в буфер.

Чтобы просмотреть каждую запись кольцевого буфера стирания, нужно открыть сеанс восстановления путем ввода команды CONTROL+Y. При этом в буфер (в текущую позицию курсора) будет скопирована самая свежая запись кольцевого буфера стирания. Если это оказалась не та запись, которая вам нужна, следует продолжить сеанс восстановления, воспользовавшись командой META+y. Это действие сотрет предыдущий восстановленный фрагмент и скопирует следующую самую свежую запись в буфер. Если окажется, что это все еще не та запись, снова введите команду META+y, чтобы стереть эту запись и извлечь копию другой записи и т. д. Так можно продолжать, пока на вернется самая старая запись. Если продолжить ввод команды META+y, то вы пройдете по кругу и вернете опять ту запись, которая была самой свежей. Таким образом, вы можете многократно изучить каждую запись.

Последовательность, используемая в сеансе восстановления, состоит из команды CONTROL+Y, за которой следует любое сочетание команд CONTROL+Y и META+y. Если после команды META+y ввести команду любого другого типа, последовательность

будет нарушена и, чтобы запустить новый сеанс восстановления, нужно будет опять воспользоваться командой CONTROL+Y.

По мере перемещения назад по кольцевому буферу стираний полезно воспринимать этот процесс как продвижение указателя последнего восстановления в глубь истории (ко все более старым записям). Этот указатель не будет переустановлен на самую свежую запись до тех пор, пока не будет дана новая команда на стирание. Используя эту технологию, можно с помощью команды CONTROL+Y и нескольких команд META+y пройти назад по кольцевому буферу стираний, ввести несколько команд, не осуществляющих стирание, а затем ввести команду CONTROL+Y и снова оказаться там, где остановился перебор элементов, продолжив его вводом последовательности команд META+y.

Можно также установить указатель последнего восстановления с помощью передачи команде META+y положительных или отрицательных аргументов. Дополнительные сведения можно найти в интерактивной документации.

Вставка специальных символов

Как ранее уже было сказано, emacs вставляет в буфер в позицию курсора все, что не является командой. Чтобы вставить символы, являющиеся командами emacs, нужно воспользоваться имеющимся в emacs символом отключения: CONTROL+Q. Этот символ можно использовать двумя способами:

- Поставить за CONTROL+Q любой другой символ, при этом данный символ вставляется в буфер, независимо от того, рассматривался он ранее как команда или нет.
- Поставить за CONTROL+Q три восьмеричных цифры, при этом байт с определяемым ими значением вставляется в буфер.

CONTROL+Q

В зависимости от настроек вашего терминала, комбинация CONTROL+Q может конфликтовать с процессом управления программным обеспечением. Если окажется, что комбинация CONTROL+Q не вызывает никакого действия, то, скорее всего, она задействована в управляющем потоке. В таком случае к команде quoted+insert нужно привязать другую клавишную последовательность (см. раздел «Изменение привязки клавиш» на с. 268).

Глобальные буферные команды

У редактора vim и его предшественников имелись глобальные команды для операций поиска и замены по всему буферу. У редактора emacs имеется похожее семейство команд. Они работают с частью буфера между указателем и концом буфера. Если нужно работать целиком со всем буфером, перед использованием команды следует воспользоваться командой META+<, чтобы переместить указатель к началу буфера.

Операции, ориентированные на работу со строками

Команды, список которых приводится в табл. 7.7, используют регулярное выражение и применяют его к строкам, находящимся между указателем и концом буфера.

Команда META+x occur помещает выводимую информацию в специальный буфер по имени *Occur*, который можно просматривать и сбрасывать или использовать в ка-

честве меню быстрых переходов на каждую строку. Чтобы использовать буфер *Occur* в качестве меню переходов, нужно переключиться на него (командой CONTROL+X o, с буквой «o» в нижнем регистре), переместить курсор на копию нужной целевой строки и ввести команду CONTROL+C CONTROL+C. Эта команда переместит курсор в буфер, в котором велся поиск, и установит его на строку, в которой было найдено соответствие регулярному выражению.

Таблица 7.7. Операции, ориентированные на работу со строками

Команда	Результат
META+x occur	Приглашение на ввод регулярного выражения и копирование каждой строки, в которой найдено соответствие регулярному выражению, в буфер по имени *Occur*
META+x delete-matching-lines	Приглашение на ввод регулярного выражения и удаление каждой строки, в которой найдено соответствие регулярному выражению
META+x delete-non-matching-lines	Приглашение на ввод регулярного выражения и удаление каждой строки, в которой не найдено соответствие регулярному выражению

Как и при любом другом изменении буфера, действие команд удаления можно отменить.

Безусловная и интерактивная замена

Команды, перечисленные в табл. 7.8, работают с символами между указателем и концом буфера, изменяя каждое соответствие строковому значению или каждое соответствие регулярному выражению. Безусловная замена все делает автоматически. При интерактивной замене вам дается возможность оценить и подтвердить каждую замену перед ее осуществлением.

Таблица 7.8. Команды замены

Команда	Результат
META+x replace-string	Приглашение на ввод *строки* и *новой_строки* и замена каждого экземпляра *строки новой_строкой*. Указатель остается в месте последней замены, но метка устанавливается в том месте, где вводилась команда, поэтому к нему можно вернуться по команде CONTROL+U CONTROL+@
META+x replace-regexp	Выводит приглашение на ввод *регулярного_выражения* и новой_строки и заменяет каждое соответствие *регулярному_выражению новой_строкой*. Указатель остается в месте последней замены, но метка устанавливается в том месте, где вводилась команда, поэтому к нему можно вернуться по команде CONTROL+U CONTROL+@
META+% *строка* или META+x query-replace	В первой форме используется *строка*, а вторая форма выводит приглашение на ввод *строки*. Обе формы выводят приглашения на ввод *новой_строки*, после чего делают запрос при встрече каждого экземпляра *строки* и, в зависимости от вашего ответа, заменяют ее *новой_строкой*. Указатель остается в месте последней замены, но метка устанавливается в том месте, где вводилась команда, поэтому к нему можно вернуться по команде CONTROL+U CONTROL+@

продолжение ⇩

Таблица 7.8 (*продолжение*)

META+x query-replace-regexp	Приглашение на ввод *регулярного_выражения* и *новой_строки*, затем запрос при встрече каждого соответствия *регулярному_выражению* и, в зависимости от вашего ответа, замена соответствующего текста *новой _строкой*. Указатель остается на месте последней замены, но метка устанавливается в том месте, где вводилась команда, поэтому к нему можно вернуться по команде CONTROL+U CONTROL+@

При осуществлении интерактивной замены emacs отображает каждый экземпляр строкового значения или соответствия регулярному выражению и спрашивает у вас, какое действие следует предпринять. В табл. 7.9 перечислены примеры возможных ответов.

Таблица 7.9. Ответы на приглашения интерактивной замены

Ответ	Значение
ВВОД	Больше не делать замен, сейчас же прекратить операцию
ПРОБЕЛ	Сделать эту замену и продолжить операци
DELETE	Не делать эту замену. Пропустить ее и продолжить операци
, (запятая)	Сделать эту замену, отобразить результат и запросить другую команду. Допустима любая команда, за исключением того, что DELETE рассматривается так же, как и ПРОБЕЛ, и не приводит к отмене изменения
. (точка)	Сделать эту замену и прекратить поиск
! (восклицательный знак)	Заменить этот и все оставшиеся экземпляры без дальнейших запросов

Посещение и сохранение файлов

Когда вы посещаете файл (такова emacs-терминология для операции «вызова файла»), emacs считывает его содержимое в буфер (см. раздел «Буферы» на с. 252), позволяя вам редактировать буфер, а затем, как правило, сохранить содержимое буфера в том же файле. Рассматриваемые здесь команды относятся к посещению и сохранению файлов.

Команды META+x pwd и META+x cd

В каждом буфере emacs хранится запись о том каталоге, который он использует по умолчанию (о каталоге, с которого был считан файл, или о рабочем каталоге, если это новый файл), стоящая впереди любого указанного вами относительного имени. Это удобное качество избавляет вас от ввода лишних данных. Чтобы вывести каталог, используемый по умолчанию для текущего буфера, следует ввести команду META+x pwd, а для вывода приглашения на ввод нового каталога по умолчанию и привязки его к данному буферу нужно ввести команду META+x cd. В следующем разделе будет рассмотрено автозаполнение имени, которым можно воспользоваться при выдаче редактором emacs приглашения на ввод имени.

Посещение файлов

Редактор emacs лучше работает в том случае, если посещается уже вызванный файл, чье отображение уже находится в буфере. После проверки времени модификации,

позволяющей убедиться, что файл не был изменен со времени своего последнего вызова, emacs просто переключается на этот буфер. В табл. 7.10 приводится перечень команд, используемых для посещения файлов.

Чтобы создать новый файл, его нужно просто вызвать. Будет создан и соответствующим образом назван пустой буфер, который впоследствии можно будет сохранить в файле. В области вывода появится сообщение о новом файле — `New File`, отражающее факт понимания ситуации со стороны emacs. Если при создании нового файла была допущена опечатка, то можно воспользоваться командой CONTROL+X CONTROL+V и ввести правильное имя.

Таблица 7.10. Посещение файлов

Команда	Результат
CONTROL+X CONTROL+F	Приглашение на ввод имени файла и считывание его содержимого в новый буфер. Присваивание буферу в качестве имени упрощенного имени файла. При этом другие буферы не затрагиваются. Обычной и зачастую полезной практикой считается иметь одновременно открытыми для редактирования сразу несколько файлов
CONTROL+X CONTROL+V	Приглашение на ввод имени файла и замена текущего буфера тем буфером, в котором находится содержимое запрошенного файла. Текущий буфер уничтожается
CONTROL+X 4 CONTROL+F	Приглашение на ввод имени файла и считывание его содержимого в новый буфер. В качестве имени буферу присваивается упрощенное имя файла. Создание нового окна для этого буфера и выбор этого окна. Окно, выбранное до ввода этой команды, по-прежнему отображает буфер, который оно показывало до этой операции, хотя новое окно может частично перекрыть старое окно

Автозавершение имени

Когда в мини-буфере появляется приглашение на ввод имени файла, можно ввести имя и нажать клавишу ВВОД. Но для помощи при вводе можно также воспользоваться функцией автозавершения имени, которая похожа на автозавершение имени файла, используемое в оболочке bash (см. раздел «Автозавершение имени» на с. 249).

В процессе ввода имени в мини-буфер нажмите клавишу TAB, и emacs, в меру своих возможностей, завершит полное имя. Если завершенное имя вас устраивает, нужно нажать ВВОД. В некоторых случаях emacs не в состоянии выполнить автозавершение. Например, введенный вами каталог может отсутствовать, или у вас может не быть прав доступа на его чтение. Если emacs не может завершить полное имя, он выводит сообщение в области вывода. Если символы, которые вы набрали за самым правым слэшем (/) в полном имени, соответствуют более чем одному имени файла, то, когда вы нажимаете клавишу TAB, emacs выводит сообщение об автозавершении имени файла, а также о том, что оно не обладает уникальностью — `[Complete, but not unique]`. Если вы нажмете клавишу TAB во второй раз, emacs откроет окно списка завершения имени, в котором будет отображен список возможных завершений (рис. 7.10). Это окно можно открыть вручную в процессе ввода полного имени, если ввести вопросительный знак (?).

При открытом окне списка завершения полного имени вы можете:

- прервать выбор, нажав комбинацию CONTROL+G или ESCAPE ESCAPE ESCAPE. Дисплей вернется в состояние, предшествовавшее открытию окна списка завершения полного имени;
- ввести еще несколько символов в мини-буфер для завершения имени. Нажать ВВОД для выбора полного имени, при этом emacs закроет окно списка завершения полного имени;
- ввести еще несколько символов в мини-буфер, чтобы добиться однозначности для автозавершения, и снова нажать клавишу TAB;
- нажать комбинацию META+v или клавишу PAGE UP, чтобы переместить курсор в окно списка завершения полного имени. Воспользоваться клавишами стрелок для перемещения курсора между вариантами выбора. Нажать ВВОД, чтобы выбрать тот вариант, на котором находится курсор. Вы можете нажать комбинацию CONTROL+G или ESCAPE ESCAPE ESCAPE, чтобы вернуться из этого окна и вернуть курсор в мини-буфер.

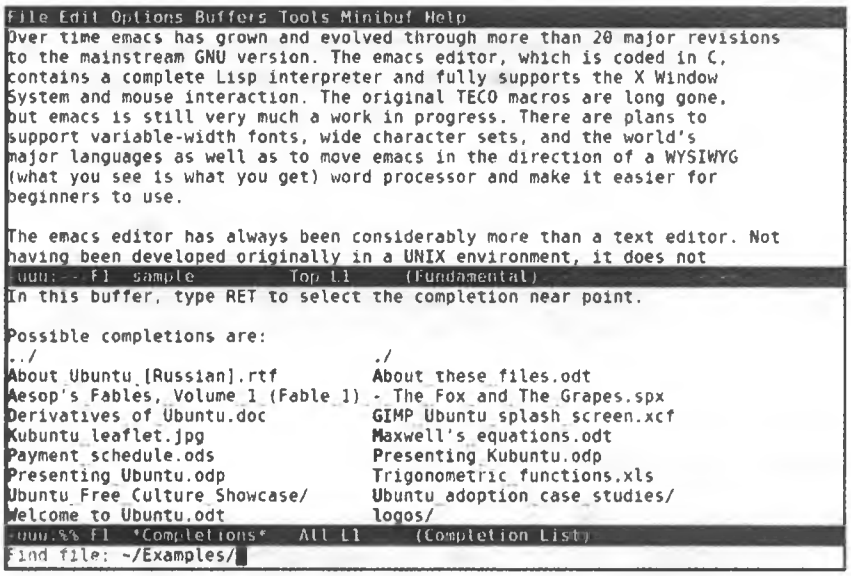

Рис. 7.10. Окно списка завершения полного имени

Когда нажимается клавиша ВВОД, emacs закрывает окно списка завершения полного имени, добавляет имя выбранного вами файла к концу набираемого имени и перемещает курсор в конец полного имени, которое вы набирали в мини-буфере. Вы можете продолжить ввод в мини-буфер и дополнить завершение перед нажатием клавиши ВВОД для принятия имени. Дополнительные сведения доступны при вызове пунктов Completion и Completion Commands меню Minibuffer интерактивного справочного руководства emacs (см. совет «Справочное руководство по emacs» на с. 225).

Сохранение файлов

Буфер сохраняется за счет копирования его содержимого в исходный, вызванный ранее файл. В табл. 7.11 перечислены связанные с этой операцией команды.

МОЖНО ВЫЙТИ БЕЗ ПОЛУЧЕНИЯ ПРЕДВАРИТЕЛЬНОГО ПРЕДУПРЕЖДЕНИЯ

Снятие флажка модификации (META+~) позволяет вам выйти не сохраняя измененный буфер без получения предупреждения. При использовании команды META+~ нужно отдавать полный отчет своим действиям.

А МОЖЕТ БЫТЬ, БУФЕР БЫЛ ИЗМЕНЕН ПО ОШИБКЕ?

Когда вы вводите команду CONTROL+X s, могут обнаружиться файлы, чьи буферы были изменены по ошибке, и может выясниться, что emacs пытается сохранить ненужные изменения в файле. Если есть неуверенность при выводе emacs-приглашения на подтверждение сохранения, не отвечайте на него утвердительно (y). Сначала выйдите из диалога, вызванного командой CONTROL+X s, набрав n для всех сохранений, в которых вы не уверены. Затем можно будет выбрать один из нескольких вариантов действий:

- Сохранить сомнительный буфер во временном файле путем ввода команды CONTROL+X CONTROL+W и проанализировать его чуть позже.
- Отменить изменения, выдавая команды CONTROL+_ до тех пор, пока из Строки режима буфера не исчезнет индикатор **.
- Если вы уверены, что все изменения ошибочны, то для получения свежей копии файла воспользоваться командой META+x revert-buffer.
- Полностью стереть буфер. Поскольку он подвергся изменениям, emacs перед выполнением этой команды сделает запрос на ее подтверждение.
- Ввести команду META+~ (тильда) для сброса состояния изменения и индикатора **. Затем ввести команду CONTROL+X s, если есть уверенность в том, что буфер не нужно записывать в файл.

Таблица 7.11. Сохранение файлов

Команда	Результат
CONTROL+X CONTROL+S	Это основная команда сохранения файла, которая сохраняет текущий буфер в его исходном файле. Если текущий буфер не подвергся изменениям, emacs выведет сообщение об отсутствии изменений, требующих сохранения — «No changes need to be saved»
CONTROL+X s	Для каждого измененного буфера будет сделан запрос на желание его сохранить. Нужно ответить положительно — «y» или отрицательно — «n». Эта команда выдается автоматически при выходе из emacs и позволяет сохранить любые буферы, подвергшиеся изменениям, но еще не записанные на диск. Эту команду следует давать для сохранения промежуточных копий вашей работы
META+x `set-visited-file-name`	Приглашение на ввод имени файла и установка его имени как «исходного» имени текущего буфера
CONTROL+X CONTROL+W	Приглашение на ввод имени файла, установка этого имени как «исходного» имени для текущего буфера и сохранение текущего буфера в файле. Является эквивалентом команды META+x `set-visited-file-name`, за которой следует команда CONTROL+X CONTROL+S
META+~ (тильда)	Сброс флажка модификации текущего буфера. Если по ошибке ввести команду META+~ для буфера с изменениями, которые требуется сохранить, то перед выходом из emacs нужно убедиться в том, что состояние изменения и его индикатор ** снова включены, иначе все внесенные изменения будут утрачены. Для того чтобы пометить буфер измененным, можно воспользоваться очень простым способом — ввести пробел, а затем удалить его с помощью клавиши DELETE

Буферы

Буфер emacs — это объект в памяти, который можно редактировать. В нем часто находится содержимое файла, но он может также существовать и без связи с файлом. Одновременно может быть выбран только один буфер, который обозначается как текущий. Большинство команд работает только с текущим буфером, даже когда окна показывают на экране несколько буферов. По большому счету, каждый буфер — это свой собственный мир: у него есть свое имя, свои собственные режимы, своя собственная связь с файлом, свое состояние измененности и, возможно, свои специальные привязки клавиш. Для создания выбора, вывода списка буферов и работы с ними можно воспользоваться командами, показанными в табл. 7.12.

Таблица 7.12. Работа с буферами

Команда	Результат
CONTROL+X b	Приглашение на ввод имени буфера и выбор этого буфера. Если буфер с таким именем отсутствует, то эта команда его создае
CONTROL+X 4 b	Приглашение на ввод имени буфера и выбор его в другом окне. Существующее окно не затрагивается, хотя новое окно может его перекрыть
CONTROL+X CONTROL+B	Создание буфера по имени *Buffer List* и вывод его в другом окне. Существующее окно не затрагивается, хотя новое окно может его перекрыть. Новый буфер не выбирается. В буфере *Buffer List* отображаются данные каждого буфера, включая имя, размер, режим (режимы) и имя исходного файла. Для буферов, открытых только для чтения, отображается знак процента (%), знак звездочки (*) показывает, что буфер изменен, а точка (.) появляется для выбора буфера
META+x rename-buffer	Приглашение на ввод нового имени буфера и присваивание этого имени текущему буферу
CONTROL+X CONTROL+Q	Переключение текущего буфера в статус и из статуса только для чтения и вывод соответствующего индикатора %% в строке режима. Эта команда может уберечь вас от случайной модификации буфера или позволить модифицировать буфер при посещении файла только для чтения
META+x append-to-buffer	Приглашение на ввод имени буфера и добавление области к концу этого буфера
META+x prepend-to-buffer	Приглашение на ввод имени буфера и добавление области в начало этого буфера
META+x copy-to-buffer	Приглашение на ввод имени буфера и удаление содержимого буфера перед копированием области в этот буфер
META+x insert-buffer	Приглашение на ввод имени буфера и вставка содержимого этого буфера в текущий буфер в текущую позицию указателя
CONTROL+X k	Приглашение на ввод имени буфера и удаление этого буфера. Если буфер подвергся изменениям, но не был сохранен, emacs запрашивает подтверждение на выполнение этой операции
META+x kill-some-buffers	Перебор списка буферов и предложение воспользоваться возможностью удалить каждый буфер. Как и в случае применения команды CONTROL+X k, emacs запрашивает подтверждение на выполнение команды стирания, если измененный буфер не был сохранен

Окна

Окно emacs является окном просмотра буфера. Экран emacs начинает работу с отображения одного окна, но позже это пространство экрана может быть поделено между двумя и более окнами. *Текущее окно* на экране содержит курсор и обеспечивает просмотр *текущего буфера*. В качестве терминологической подсказки воспользуйтесь советом «Экран и окна emacs» на с. 222.

Команда CONTROL+X b имя_буфера

Окно в каждый момент времени отображает содержимое только одного буфера. Команда CONTROL+X b *имя_буфера* переключает буфер, отображаемый в текущем окне. Один и тот же буфер может отображаться в нескольких окнах, причем в разных окнах могут отображаться различные части этого буфера. Любое изменение буфера сказывается на всех окнах, отображающих этот буфер. Кроме того, буфер может существовать и без открытого для него окна.

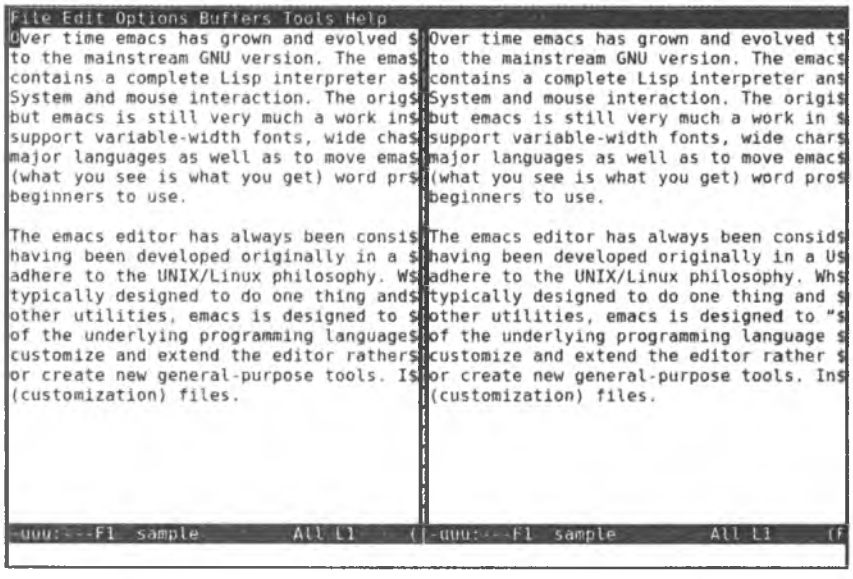

Рис. 7.11. Горизонтальное разбиение окна

Разбиение окна

Одним из способов деления окна является разбиение стартового окна на две и более частей. Команда CONTROL+X 2 разбивает текущее окно на два, расположенных друг над другом. Числовой аргумент используется в качестве размера верхнего окна в строках. Команда CONTROL+X 3 разбивает текущее окно на два, размещенных рядом (рис. 7.11). Числовой аргумент взят в качестве количества столбцов, выделяемых левому окну. К примеру, команда CONTROL+U CONTROL+X 2 разбивает текущее окно на два, поскольку команда CONTROL+U со специальной интерпретацией «умножить на 4» стоит обособленно, верхнее окно получает четыре строки (минимально достаточное количество для работы).

Хотя эти команды разбивают текущее окно, в обоих окнах продолжается просмотр одного и того же буфера. Для любого из окон (или для обоих сразу) можно выбрать новый буфер или можно прокрутить каждое окно, чтобы в нем отображался другой фрагмент того же буфера.

Работа с окнами

Команды CONTROL+X o и META+ CONTROL+V

Для выбора другого окна можн воспользоваться командой CONTROL+X o (буква «о» в нижнем регистре). Если на экране появляется более двух окон, то последовательность команд CONTROL+X o обеспечивает переход между ними по кругу, в направлении сверху вниз и слева направо. Команда META+CONTROL+V прокручивает другое окно. Если отображается более двух окон, команда прокручивает то окно, которое будет выбрано следующим командой CONTROL+X o. Можно использовать как положительный, так и отрицательный аргумент прокрутки, точно так же, как и при использовании команды CONTROL+V для прокрутки текущего окна.

Отображение дополнительных окон

Команды CONTROL+X 4b и CONTROL+X 4f

В обычных операциях emacs явное разбиение окна осуществляется так же редко, как и неявное разбиение, проводимое с помощью семейства команд CONTROL+X 4. К примеру, команда CONTROL+X 4b выводит приглашение на ввод имени буфера и выбирает его в другом окне. Если не существует никаких других окон, эта команда начинает с разбиения окна на две равных половины и размещения окон одно над другим. Команда CONTROL+X 4f выводит приглашение на ввод имени файла, вызывает этот файл в другом окне и выбирает это окно. Если не существует никаких других окон, эта команда начинает с разбиения окна на две равных половины и размещения окон одно над другим.

Настройка и удаление окон

Команды CONTROL+X 0 и CONTROL+ X 1

Когда окна мешают, их можно удалить. При этом в буфере, который связан с окном, данные не теряются, и как только понадобится, можно будет создать другое окно. Команда CONTROL+X 0 (нуль) удаляет текущее окно и передает его пространство соседнему окну, а команда CONTROL+X 1 удаляет все окна за исключением текущего окна.

Команды META+x shrink-window, CONTROL+X ^, CONTROL+X } и CONTROL+ X {

Можно также настроить размеры текущего окна за счет его соседей. Чтобы сделать окно короче, нужно ввести команду META+x shrink-window. Чтобы увеличить высоту окна, нужно ввести команду CONTROL+X ^, а чтобы сделать его шире, нужно ввести команду CONTROL+X }, сузить окно можно с помощью команды CONTROL+X {. Каждая из этих команд добавляет к окну или вычитает из него одно строку или один столбец, если только перед командой не был введен числовой аргумент.

У редактора emacs существуют свои нормативы для минимально полезного размера окна, и он может уничтожить окно еще до того, как вы сведете один из его размеров к нулю. Хотя окно может исчезнуть, но буфер при этом останется нетронутым.

Команды оболочки, выполняемые в активном режиме

Редактор emacs может запустить подчиненную оболочку (оболочку, запускаемую в качестве дочерней программы emacs; см. раздел «Выполнение команды» на с. 223) для выполнения одной командной строки с необязательным стандартным вводом

из области текущего буфера и необязательным стандартным выводом, заменяющим область (табл. 7.13). Этот процесс аналогичен выполнению команды оболочки из редактора vim, где стандартный ввод исходил из редактируемого файла, а стандартный вывод возвращался в тот же файл (см. раздел «Команда :!!команда» на с. 210). Успешная работа этого процесса зависит, как и при работе с редактором vim, от возможностей оболочки.

Таблица 7.13. Запуск команд оболочки в активном режиме

Команда	Результат
META+! (восклицательный знак)	Приглашение на ввод команды оболочки, выполнение этой команды и отображение выходной информации
CONTROL+U META+! (восклицательный знак)	Приглашение на ввод команды оболочки, выполнение этой команды и вставка выходной информации в место, где находится указатель
META+| (вертикальная черта)	Приглашение на ввод команды оболочки, предоставление ей области в качестве ввода, фильтрация ее через команду и отображение выходной информации команды
CONTROL+U META+| (вертикальная черта)	Приглашение на ввод команды оболочки, предоставление ей области в качестве ввода, фильтрация ее через команду, удаление старой области и вставка с этой позиции выходной информации команды

Редактор emacs может также запускать интерактивную подчиненную оболочку, которая запускается на постоянной основе в своем собственном буфере. Дополнительные сведения даны в разделе «Режим оболочки» на с. 265.

Команды оболочки, выполняемые в фоновом режиме

Редактор emacs может запускать процессы в фоновом режиме, с выводом, направляемым в наращиваемый буфер emacs, который не должен оставаться в поле зрения. При работе фонового процесса можно продолжать редактирование и посмотреть на выведенную им информацию чуть позже. В этом режиме может быть запущена любая команда оболочки.

Наращиваемый выходной буфер всегда называется *compilation*. Его можно читать, из него можно копировать данные или его можно редактировать любым способом, не дожидаясь окончания фонового процесса. Чаще всего этот буфер используется для просмотра выходной информации при компиляции программы и для исправления любых синтаксических ошибок, найденных компилятором.

Команда META+x compile

Чтобы запустить процесс в фоновом режиме, нужно ввести команду META+x compile, которая выведет приглашение на ввод команды оболочки и начнет выполнять ее в виде фонового процесса. Для отображения буфера *compilation* экран разбивается на две половины.

Если нужно, то можно переключиться на буфер *compilation* и понаблюдать за выполнением. Чтобы прокручивать дисплей в процессе этого наблюдения, нужно поставить курсор в самый конец текста, воспользовавшись командой META+>. Если вас не интересует это отображение, то просто удалите окно командой CONTROL+X 0 (нуль),

если вы в нем находитесь, или, в противном случае, командой CONTROL+X 1 и продолжайте работать. Позже можно будет снова переключиться на буфер *compilation*, воспользовавшись командой CONTROL+X b.

Чтобы уничтожить фоновый процесс, нужно ввести команду META+x `kill-compilation`. Редактор emacs запросит подтверждение, а затем уничтожит фоновый процесс.

Если в буфере *compilation* появятся сообщения об ошибках стандартного формата, вы сможете автоматически посетить ту строку в файле, где произошла ошибка. Чтобы разбить экран на два окна и посетить файл и строку, фигурирующую в следующем сообщении об ошибке, введите команду CONTROL+X‘ (обратный апостроф). Прокрутите буфер *compilation* до появления этого сообщения об ошибке в верхней части его окна. Чтобы начать с первого сообщения об ошибке и посетить файл и строку, воспользуйтесь командой CONTROL+U CONTROL+X‘.

Основные режимы: редактирование с восприимчивостью к языкам программирования

Редактор emacs обладает большой коллекцией наборов свойств, и каждый из этих наборов предназначен для конкретной разновидности текста. Наборы свойств называются *основными режимами*. Буфер может одновременно иметь только один основной режим.

Основной режим буфера является закрытым и не влияет на редактирование в любом другом буфере. При переключении на новый буфер, имеющий другой режим, правила для нового режима вступают в силу немедленно. Чтобы избежать недоразумений, имя основного режима буфера появляется в строке режима любого окна, в котором ведется просмотр этого буфера (см. рис. 7.1).

Существуют три класса основных режимов, которые используются для следующих задач:

- Редактирования текста на простом человеческом языке (например, text, nroff, TeX).
- Редактирования кода, написанного на языках программирования (например, C, Fortran, Lisp).
- Выполнения задач специального назначения (например, shell, mail, dired, ftp).

Кроме этого, существует еще один основной режим — Fundamental, который не делает ничего особенного. Основной режим обычно определяет следующие установки:

- Специальные команды, уникальные для режима, возможно, со своими собственными клавиатурными привязками. У языковых режимов может быть всего несколько специальных команд, а у режимов специального назначения их могут быть десятки.
- Специфичный для режима символьный синтаксис и регулярные выражения, определяющие символы, составляющие слова, разделители, комментарии, пробельные символы и т. д. Эта установка обуславливает поведение команд, ориентированных на синтаксические единицы, такие как слова, предложения, комментарии или выражения, заключенные в скобки.

Выбор основного режима

Команда META+x имя_режима

Редактор emacs выбирает и устанавливает режим, когда файл вызывается на редактирование. Для этого он сравнивает имя файла с определенными регулярными выражениями, описывающими имя файла и его расширение. Непосредственная установка основного режима осуществляется с помощью команды META+x *имя_режима*. Эта команда используется в основном для исправления основного режима, когда «догадки» редактора emacs оказываются неверными.

Чтобы файл определял свой собственный режим, в какое-нибудь место первой непустой строки файла нужно включить текст -*- *имя_режима* -*-, возможно, внутри комментария, подходящего для того языка программирования, на котором написан содержащийся в файле код.

Режимы «человеческого» языка

Человеческий язык предназначается в конечном итоге для использования людьми, возможно, после того, как он пройдет обработку в программе форматирования текста. Человеческие языки придерживаются многих соглашений о структуре слов, предложений и абзацев. По отношению к этим текстовым элементам все основные режимы человеческого языка ведут себя одинаково.

За пределами этой области общих представлений каждый режим предлагает дополнительные функциональные возможности, ориентированные на определенное средство форматирования текста, например TeX, LaTeX или nroff/troff. Расширения, занимающиеся форматированием текста, не относятся к теме данной главы, главное внимание которой сосредоточено на командах, имеющих отношение к текстовым элементам человеческого языка.

Слова

Чтобы команды было легче запоминать, привязки клавишных последовательностей к командам, работающим со словами, перекликаются с привязками к командам, работающим с символами: CONTROL+F, CONTROL+B, CONTROL+D, DELETE и CONTROL+T.

Команды META+f и META+b

Точно так же, как команды CONTROL+F и CONTROL+B перемещают курсор вперед и назад по символам, команды META+f и META+b перемещают его вперед и назад по словам. Они могут приступать к работе из позиции внутри или за пределами слова, по которому проходит курсор, но во всех случаях указатель останавливается вне слова, примыкая к последнему пройденному символу. Обе команды воспринимают числовой аргумент, определяющий количество слов, через которое проходит указатель.

Команды META+d и META+DELETE

По аналогии с тем, как комбинация CONTROL+D и клавиша DELETE удаляют символы в прямом и обратном направлении, комбинации META+d и META+DELETE стирают слова вперед и назад. Они оставляют указатель на тех же самых финишных позициях, на которых его оставляли команды META+f и META+b, но при этом они стирают слова, через которые проходят. Эти команды также воспринимают числовой аргумент.

Команда META+t

Команда META+t меняет местами слово до указателя со словом после него.

Предложения

Команды META+a, META+e, CONTROL+X DELETE и META+k

Чтобы команды было легче запоминать, три привязки клавишных последовательностей к командам, работающим с предложениями, перекликаются с привязками к командам, работающим со строками: CONTROL+A, CONTROL+E и CONTROL+K. Команда META+a перемещает курсор назад к началу предложения, команда META+e перемещает курсор вперед к концу предложения. В дополнение к ним, команда CONTROL+X DELETE стирает предложение в обратном направлении до его начала, а команда META+k стирает предложение в прямом направлении до его конца.

Редактор emacs распознает концы предложений, обращаясь к регулярному выражению, которое содержится в переменной по имени sentence-end. Короче говоря, emacs ищет символы ., ? или !, за которыми следует два пробела или признак конца строки, возможно, с закрывающими кавычками или закрывающими скобками. Чтобы отобразить значение этой переменной, нужно ввести команду CONTROL+H v sentence-end ВВОД.

Команды META+a и META+e оставляют указатель примыкающим к первому или непустому символу в предложении. Они воспринимают числовой аргумент, определяющий количество предложений, через которое проходит указатель, отрицательный аргумент запускает их в обратном направлении.

Команды META+k и CONTROL+X DELETE стирают предложения вперед и назад, точно так же, как команда CONTROL+K стирает строку. Эти команды оставляют указатель в том же самом месте, где его оставляют команды META+a и META+e, но они стирают те предложения, через которые проходят. Они также воспринимают числовой аргумент. Команда CONTROL+X DELETE хорошо подходит для быстрой отмены частично набранного предложения.

Абзацы

Команды META+{, META+} и META+h

Команда META+{ перемещает курсор назад на начало самого последнего пройденного абзаца, а команда META+} перемещает курсор вперед на конец следующего абзаца. Команда META+h помечает абзац, на котором находится курсор, как область (то есть она помещает указатель на начало, а метку — на конец этого абзаца), или помечает следующий абзац, если курсор находится между абзацами.

Команды META+} и META+{ оставляют указатель в начале строки, примыкающей соответственно к первому или последнему символу абзаца. Эти команды воспринимают числовой аргумент, определяющий количество абзацев, которые нужно пройти, и работают в обратном направлении, если им дается отрицательный аргумент.

В режимах, ориентированных на простой человеческий язык, абзацы отделены пустыми строками и строками, содержащими команды форматирования текста, а сам абзац начинается со строки, имеющей отступ. Распознавание абзаца основывается на регулярных выражениях, хранящихся в переменных paragraph-separate и paragraph-start. Абзац состоит из завершенных строк, включающих завершающий признак конца строки. Если абзац начинается после одной или нескольких пустых строк, то последняя пустая строка перед абзацем принадлежит этому абзацу.

Заполнение

Редактор emacs может заполнять абзац, чтобы он занимал указанную ширину, разбивая строки и реконфигурируя их по мере необходимости. Разбиение строк осуществляется между словами, без осуществления переноса слов. Редактор emacs может проводить автоматическое заполнение по мере ввода символов или в ответ на конкретную команду.

Команда META+x auto-fill-mode

Команда META+x `auto-fill-mode` включает и выключает режим Автозаполнения (Auto Fill). Когда этот режим включен, emacs автоматически разбивает строки, когда вы вводите пробел или нажимаете клавишу ВВОД и оказываетесь за пределами указанной ширины строки. Это свойство полезно при вводе нового текста.

Команды META+q и META+x fill-region

Режим Автозаполнения не занимается автоматическим перезаполнением всего абзаца, с которым вы работаете в данный момент. Если текст добавляется в середине абзаца, режим Автозаполнения разбивает новый текст по мере его ввода, но не занимается перезаполнением всего абзаца. Чтобы провести перезаполнение абзаца или области абзаца, нужно использовать либо команду META+q для перезаполнения текущего абзаца, либо команду META+x `fill-region` для перезаполнения каждого абзаца в области (между указателем и меткой).

Можно изменить установленную по умолчанию ширину заполнения, имеющую значение 70, присвоив новое значение переменной `fill-columna`. Чтобы установить значение переменной `fill-column` на текущую позицию курсора, нужно ввести команду CONTROL+X f, а чтобы установить значение переменной `fill-column` равным числу *nnn*, где число 0 указывает на левую границу, нужно ввести команду CONTROL+U *nnn* CONTROL+X f.

Изменение регистра букв

Редактор emacs может привести все символы в словах или областях к верхнему, нижнему регистру или к регистру с первыми заглавными буквами (при котором первая буква каждого слова переводится в верхний регистр, а все остальные буквы — в нижний регистр) (табл. 7.14).

Таблица 7.14. Изменение регистра букв

Команда	Результат
META+l (буква «l» в нижнем регистре)	Перевод символов слова, находящегося справа от указателя, в нижний регистр
META+u	Перевод символов слова, находящегося справа от указателя, в верхний регистр
META+c	Перевод символов слова, находящегося справа от указателя, в регистр с первыми заглавными буквами
CONTROL+X CONTROL+L	Перевод символов области в нижний регистр
CONTROL+X CONTROL+U	Перевод символов области в верхний регистр

Изменение, касающееся слова, перемещает указатель по этому слову (точно так же, как это делает команда META+f), позволяя перемещаться по тексту и изменять символы каждого слова с помощью команды META+l, META+u или META+c или оставлять их без изменений, переходя дальше с помощью команды META+f. Положительный числовой аргумент приводит к изменению указанного количества слов справа от указателя, перемещая указатель по мере изменения.

Отрицательный числовой аргумент изменяет указанное количество слов слева от указателя, но оставляет указатель на прежнем месте. Это свойство можно применить для быстрого изменения регистра символов в только что введенных словах. В табл. 7.15 показаны несколько примеров использования этих команд.

Таблица 7.15. Примеры изменения регистра

Символы и команды	Результат
HELLOMETA+–META+l (буква «l» в нижнем регистре)	Hello
helloMETA+ –META+u	HELLO
helloMETA+ –META+c	Hello

Если курсор (указатель) находится в середине слова, команды изменения регистра воздействуют на символы, расположенные слева от курсора.

Текстовый режим

Команда META+x text-mode

За редким исключением, команды для текстовых элементов, связанных с простым человеческим языком, всегда включены и доступны, даже когда активированы режимы, связанные с языками программирования. Текстовый режим является совсем небольшим добавлением к основным командам, но его все же лучше включить, хотя бы для активации функции клавиши TAB (см. ниже). Для активации текстового режима нужно воспользоваться командой META+x `text-mode`.

Команда META+x edit-tab-stops

Нажатие клавиши TAB в текстовом режиме запускает функцию tab-to-tab-stop. Согласно настройкам по умолчанию, TAB останавливает курсор через каждые восемь столбцов. Эти настройки можно изменить с помощью команды META+x `edit-tab-stops`, которая вызывает переход к специальному буферу *`Tab Stops`*. В этом буфере дается раскладка текущих остановок табуляции в том масштабе, в котором проводится редактирование. Новые остановки устанавливаются, когда вводится команда CONTROL+C CONTROL+C. Этот буфер можно стереть (CONTROL+X k) или из него можно переключиться на другой буфер (CONTROL+X b), не меняя при этом остановок при нажатии клавиши TAB.

Остановки табуляции, установленные с помощью команды META+x `edit-tab-stops`, влияют только на интерпретацию символов табуляции, поступающих с клавиатуры. Редактор emacs автоматически вставляет количество пробелов, достаточное для достижения остановки табуляции. Эта команда не влияет на интерпретацию символов табуляции, уже находящихся в буфере или в исходном файле. Если остановки табуляции редактируются, а затем используются, то при распечатке файла на бумаге текст будет выглядеть точно так же, как на экране.

Режим языка C

Код, написанный на языках программирования, читается людьми, но интерпретируется машинами. Кроме продолжения обработки некоторых текстовых элементов простого человеческого языка (например, слов и предложений), основные режимы работы с языками программирования обращаются к решению нескольких дополнительных задач:

- Обработка *сбалансированных выражений*, заключенных в круглые, квадратные или фигурные скобки как текстовых модулей.
- Обработка комментариев как текстовых элементов.
- Введение отступов.

В редактор emacs включены основные режимы для поддержки C, Фортрана и нескольких вариантов Лиспа. Кроме этого, многие пользователи поспособствовали появлению режимов для своих любимых языков. В этих режимах по-прежнему доступны команды для человеческих текстовых элементов, но иногда уже в переопределенном виде. Например, абзац ограничен только пустыми строками, а отступ не является признаком начала абзаца. Кроме этого, каждый режим содержит специализированный код для учета соглашений, специфичных для данного языка, которые касаются сбалансированных выражений, комментариев и отступов. В этой главе рассматривается только режим языка C.

Выражения

Основные режимы emacs обладают ограниченными возможностями по лексическому анализу. Они в состоянии распознать большинство лексем (например, символы, строки и числа) и все парные наборы круглых, квадратных и фигурных скобок. Этого достаточно для Лиспа, но не для C. Режим языка C не имеет полнофункционального синтаксического анализатора и не готов к распознаванию всевозможных выражений на языке C[1].

В табл. 7.16 перечислены команды emacs, применимые к выражениям, заключенным в скобки, и к некоторым лексемам. По своей конструкции привязки перекликаются с командами CONTROL для символов и командами META для слов. Все эти команды воспринимают числовой аргумент и запускаются в обратном направлении, если он имеет отрицательное значение.

Таблица 7.16. Команды для выражений и лексем

Команда	Результат
CONTROL+META+f	Перемещение указателя вперед на одно выражение. Конкретное поведение зависит от того, какой символ находится справа от указателя (или слева от указателя, в зависимости от направления его перемещения). • Если первым непустым символом является открывающий ограничитель (круглая, квадратная или фигурная скобка), указатель перемещается прямо за соответствующий закрывающий ограничитель. • Если первое непустое пространство занимает лексема, указатель перемещается сразу же за нее

продолжение

[1] В документации по emacs периодически встречающийся термин sexp относится к термину Лиспа S-expression. К сожалению, он иногда бессистемно используется и для выражения (expression), даже если не относится к языку Лисп.

Таблица 7.16 (*продолжение*)

Команда	Результат
CONTROL+META+b	Перемещение указателя назад на одно выражение
CONTROL+META+k	Стирание одного, находящегося впереди выражения. Эта команда оставляет указатель в той же финишной позиции, что и команда CONTROL+META+f, но стирает выражение, через которое он проходит
CONTROL+META+@	Установка метки в той позиции, куда бы осуществила перемещение команда CONTROL+META+f, но без изменения позиции указателя. Чтобы отчетливо увидеть помеченную область, нужно ввести пару команд CONTROL+X CONTROL+X для смены местами указателя и метки

Определение функции

При работе с emacs сбалансированное выражение, находящееся ближе всего к левому краю экрана, считается определением функции и часто называется *defun*, хотя этот термин более характерен для Лиспа. В более общем смысле под ним понимается определение функции на том языке, с которым ведется работа.

В режиме языка C определение функции включает возвращаемый тип данных, имя функции и объявление аргумента, появляющееся перед символом {. В табл. 7.17 показаны команды для работы с определениями функций.

СТИЛЬ ОТСТУПА В ФУНКЦИИ

В редакторе emacs открывающая фигурная скобка с левого края считается частью определения функции. Это ускоряет обратное сканирование, производящееся для определения начального элемента. Если в вашем коде используется стиль отступа, при котором открывающая фигурная скобка помещается в каком-нибудь другом месте, у вас могут быть получены весьма неожиданные результаты.

Таблица 7.17. Команды определения функции

Команда	Результат
CONTROL+META+a	Перемещение указателя к началу самого последнего определения функции. Эта команда используется для обратного сканирования буфера для поиска очередного предыдущего определения функции
CONTROL+META+e	Перемещение указателя к концу следующего определения функции. Эта команда используется для сканирования буфера в прямом направлении для поиска очередного определения функции
CONTROL+META+h	Пометка как области текущего определения функции (или следующего определения функции, если курсор находится между двумя определениями). Эта команда настраивает все определение функции для работы с ним как с областью, например, для того чтобы его стереть

Отступы

Используемый в emacs режим языка C обладает развитой логикой для управления отступами в программе на этом языке. Эту логику можно настроить под многие другие стили отступов в программе, написанной на языке C (табл. 7.18).

Таблица 7.18. Команды отступов

Команда	Результат
TAB	Настройка отступа текущей строки. Команда TAB вставляет или удаляет пробелы в начале строки, пока отступ не будет приведен в соответствие с текущим контекстом и действующими правилами. Указатель не перемещается, если только он не находится в пробельном пространстве; в таком случае он перемещается к концу этого пространства. Команда TAB вставляет лидирующие пробелы; клавишу TAB можно нажимать при любой позиции курсора на строке. Если нужно вставить в текст символ табуляции, следует воспользоваться комбинацией META+i или CONTROL+Q TAB
LINEFEED (Перевод строки)	Сокращенная форма для команды ВВОД, за которой следует команда TAB. Клавишу LINEFEED удобно использовать для ввода нового кода, при котором получается автоматический отступ при каждом переходе на новую строку
Следующие две команды создают при вводе одной команды отступы сразу на нескольких строках	
CONTROL+META+q	Создание заново всех отступов внутри следующих парных фигурных скобок. Команда CONTROL+META+q предполагает, что у левой фигурной скобки имеется правильный отступ, и управляет отступом с ее позиции. Если нужно настроить позицию левой фигурной скобки, то перед вводом этой команды следует нажать TAB, установив указатель левее этой скобки. Все строки, вплоть до относящейся к этой паре закрывающей фигурной скобки, получат отступы, как будто для каждой из них была применена команда TAB
CONTROL+META+\	Создание заново всех отступов в области. Нужно установить указатель левее левой фигурной скобки, а затем ввести эту команду. Все строки, вплоть до соответствующей закрывающей фигурной скобки, получат отступы, как будто для каждой из них была применена команда TAB

Настройка отступов

Со временем возникло множество стилей программирования на языке C, и в emacs делается все для обеспечения автоматических отступов для всех этих стилей. Для emacs версии 19 режим ввода кода с отступами был полностью переписан и поддерживает C, C++, Objective-C и Java. Новые синтаксические анализаторы emacs работают намного точнее и способны классифицировать каждый синтаксический элемент каждой строки программы, относя его к определенной синтаксической категории (число которых приближается к 50), например к *инструкции*, *строке* или к оператору *else*. На основе этого анализа emacs обращается к таблице смещений по имени `c-offsets-alist`, чтобы найти величину отступа каждой строки от предыдущей.

Для настройки отступов нужно изменить таблицу смещений. Хотя можно определить совершенно новую таблицу смещений для каждого настраиваемого стиля, обычно удобнее ввести небольшой список исключений в стандартные правила. Такой список исключений имеется в `c-style-alist` для каждого основного стиля (GNU, K&R [Kernighan and Ritchie], BSD и т. д.). Вот как выглядит одна из записей таблицы `c-style-alist`:

```
("gnu"
(c-basic-offset . 2)
```

```
(c-comment-only-line-offset . (0 . 0))
(c-offsets-alist . ((statement-block-intro . +)
    (knr-argdecl-intro . 5)
    (substatement-open . +)
    (label . 0)
    (statement-case-open . +)
    (statement-cont . +)
    (arglist-intro . c-lineup-arglist-intro-after-paren)
    (arglist-close . c-lineup-arglist)
    ))
)
```

Вопросы создания пользовательских стилей не входят в тему данной книги. Если интересно, то в info-страницах по emacs можно найти много сведений на эту тему, начиная с раздела «Customizing C Indentation» («Настройка отступов для C»). Образец файла .emacs, приведенный далее в данной главе, добавляет очень простой пользовательский стиль и приспосабливает его для использования с каждым редактируемым файлом, имеющим расширение .c.

Комментарии

У каждого буфера имеется своя собственная переменная `comment-column`, которую можно просмотреть с помощью команды справки CONTROL+H v `comment-column` ВВОД. В табл. 7.19 приведен список, облегчающий работу с комментариями.

Таблица 7.19. Команды для работы с комментариями

Команда	Результат
META+;	Вставка комментария в текущую строку или выравнивание уже существующего комментария. Поведение команды зависит от конкретной ситуации. ▪ Если в данной строке нет комментария, META+; создает пустой комментарий, используя значение переменной `comment-column`. ▪ Если текст, уже имеющийся в данной строке, перекрывает позицию, указанную в переменной `comment-column`, команда META+; создает пустой комментарий на один пробел после конца строки. ▪ Если комментарий в данной строке уже имеется, но размещен не в соответствии с текущим значением переменной `comment-column`, команда META+; помещает комментарий в указанный столбец. Если этому мешает текст, она помещает комментарий на один пробел после конца текста. Как только выровненный (возможно пустой) комментарий будет присутствовать в строке, указатель перемещается в начало комментируемого текста
CONTROL+X ;	Установка значения переменной `comment-column` на столбец после указателя. Левый край является столбцом 0
CONTROL+U — CONTROL+X ;	Стирание комментария в текущей строке. Эта команда устанавливает значение `comment+column` из первого комментария, найденного выше данной строки, а затем выполняет команду META+;, чтобы вставить или выровнять комментарий в этой позиции
CONTROL+U CONTROL+X ;	Установка значения `comment-column` на позицию первого найденного выше данной строки комментария, а затем выполнение команды META+; для вставки или выравнивания комментария в данной строке

Режимы специального назначения

У редактора emacs есть еще третье семейство Основных режимов, которое не ориентировано ни на конкретный язык, ни на конкретное обычное редактирование. Эти режимы выполняют некую специальную функцию. Для выполнения этой функции могут определяться следующие режимы со своими собственными клавишными привязками и командами:

- Rmail: читает, архивирует и составляет сообщения электронной почты..
- Dired: обходит вывод информации командой `ls -l` на дисплей и работает с файлами.
- VIP: имитирует полную среду окружения редактора vi.
- VC: позволяет управлять системами управления версиями (включая RCS, CVS и Subversion) прямо из emacs.
- GUD (Grand Unified Debugger — большой унифицированный отладчик): позволяет запускать и отлаживать программы на языке C (и на других языках) прямо из emacs.
- Tramp: позволяет редактировать файлы на любой удаленной системе, к которой можно получить доступ через ftp или scp.
- Shell: запускает интерактивную оболочку прямо из буфера emacs.

  В данной книге рассматривается только режим оболочки — Shell mode.

Режим оболочки

Выполнение одиночных команд оболочки и применение фильтра к области были рассмотрены ранее в разделе «Команды оболочки, выполняемые в активном режиме» на с. 254. Каждый буфер emacs в Режиме оболочки имеет постоянно связанную с ним базовую интерактивную оболочку. Эта оболочка получает свой ввод из последней строки буфера и отправляет свой вывод обратно в буфер, перемещая указатель по мере вывода. Если не редактировать буфер, то он будет содержать запись всего сеанса работы с оболочкой.

Оболочка запускается в асинхронном режиме, независимо от того, просматриваете вы ее буфер или нет. Для чтения вывода оболочки и добавления его в буфер редактор emacs использует время простоя.

Команда META+x shell

Для создания буфера по имени `*shell*` и запуска подчиненной оболочки следует ввести команду **META+x** `shell`. Если буфер с именем `*shell*` уже существует, emacs просто переключается на этот буфер. Имя оболочки, запускаемой этой командой, берется из одного из следующих источников:

- Лисп-переменной `explicit-shell-file-name`.
- Переменной окружения `ESHELL`.
- Переменной окружения `SHELL`.

Чтобы запустить вторую оболочку, сначала нужно ввести команду **META+x** `rename-buffer` для изменения имени существующего буфера оболочки, а затем ввести команду **META+x** `shell` для запуска новой оболочки. Можно создать столько угодно подчиненных оболочек и буферов, работающих параллельно.

В Режиме оболочки определяется специальный набор команд (табл. 7.20). Эти команды привязаны в основном к двухклавишным последовательностям, которые начинаются с комбинации CONTROL+C. Каждая последовательность похожа на обычные символы управления, имеющиеся в Linux, но использует лидирующую комбинацию CONTROL+C.

Таблица 7.20. Команды Режима оболочки

Команда	Результат
ВВОД	Если указатель находится в конце буфера, emacs вставляет символ, генерируемый клавишей ВВОД, и отправляет эту (последнюю) строку оболочке. Если указатель находится где-нибудь в другом месте, команда копирует строку, в которой он находится, в конце буфера, очищая старое приглашение оболочки (см. регулярное выражение shell-prompt-pattern), если таковое существует. Затем эта скопированная строка, которая теперь является последней строкой буфера, отправляется оболочке
CONTROL+C CONTROL+D	Отправка символа CONTROL+D оболочке или подчиненной ей оболочке
CONTROL+C CONTROL+C	Отправка символа CONTROL+C оболочке или подчиненной ей оболочке
CONTROL+C CONTROL+\	Отправка сигнала завершения сеанса — quit, оболочке или подчиненной ей оболочке
CONTROL+C CONTROL+U	Стирание текста в текущей незавершенной строке
CONTROL+C CONTROL+R	Обратная прокрутка к началу последней, выведенной оболочкой информации, помещение первой строки выведенной информации в верхнюю часть окна
CONTROL+C CONTROL+O	Удаление последнего пакета информации, выведенной оболочкой

Настройка emacs

В основе emacs лежит Лисп-интерпретатор, написанный на языке C. Эта версия Лиспа существенно расширена и включает множество специальных команд редактирования. Главная задача интерпретатора заключается в выполнении систем, запрограммированных на Лиспе, которые реализуют внешний вид и функциональность emacs.

Сокращенная до самого необходимого, эта система реализует непрерывный цикл, отслеживающий нажатия клавиш, проводящий их разбор и превращение в команды, выполняющий эти команды и обновляющий изображение на экране. Для настройки этого режима работы можно воспользоваться несколькими способами.

- Как только нажата отдельная комбинация клавиш, тут же по таблице перевода определяется, к чему она привязана. Изменяя записи в этой таблице, можно менять назначение клавиш. Если, к примеру, вы привыкли работать в редакторе vi или vim, то может появиться желание изменить назначение клавиш DELETE и CONTROL+H, чтобы комбинация CONTROL+H вызывала забой, как она это делает в vim, а клавиша DELETE (которая не используется в vim) была справочной клавишей. Если клавиша DELETE используется в качестве клавиши прерывания, то для обмена значениями с CONTROL+H может понадобиться выбрать другую клавишу.
- Коды нажатых клавиш, отображаемые на команды, собраны в небольшую группу, называемую *клавишными последовательностями*. В качестве клавишной последовательности может выступать одна клавишная комбинация, например CONTROL+N, или две и более клавишных

комбинаций, например CONTROL+X CONTROL+F. В сгруппированном виде клавишная последовательность используется для выбора конкретной выполняемой процедуры. Правила группировки каждой клавишной последовательности и конкретное имя процедуры, выполняемой при поступлении кодов этой клавишной последовательности, закодированы в серии таблиц, называемых *картами привязки*. Изменяя их, можно менять правила группировки или систему привязки процедур к клавишным последовательностям. Например, если вы привыкли к комбинации CONTROL+W, используемой в редакторах vi и vim для забоя вводимого слова, то может появиться желание изменить существующую в emacs привязку комбинации CONTROL+W со стандартного стирания области (`kill-region`) на удаление слова в обратном направлении (`delete-word-backward`).

- Работа команд часто зависит от значений глобальных переменных или ключей. Можно достичь нужных характеристик путем настройки некоторых из этих переменных.
- Обычно сама команда представляет собой Лисп-программу, которую можно перепрограммировать, чтобы заставить ее работать нужным образом. Хотя подобная задача не по плечу новичкам, исходный Лисп-код практически всех команд имеется в свободном доступе, а встроенная Лисп-система полностью документирована. Как уже ранее упоминалось, по установившейся практике настроенный Лисп-код загружается во время запуска редактора, даже если вы не создавали его самостоятельно.

В большей части документации по emacs этапы перевода, группировки и выбора процедуры замалчиваются и о нажатиях клавиш говорится как о командах. Тем не менее важно осознавать наличие лежащего в основе всего этого механизма и понимать, что его поведение можно изменить.

Файл запуска .emacs

При каждом запуске emacs загружается файл ~/.emacs, в котором содержится Лисп-код. Чаще всего для настройки emacs используется именно этот файл. Использование файла .emacs управляется двумя ключами командной строки. Ключ `-q` приводит к игнорированию файла .emacs, в результате чего редактор emacs запускается без него. Это один из способов обхода неподходящего файла .emacs. Ключ `-u` (user — пользовательский) приводит к использованию файла ~user/.emacs (файла .emacs из личного каталога *пользователя*).

Файл запуска .emacs обычно имеет отношение только к клавишным привязкам и установкам ключей. Для него можно писать инструкции на Лиспе в непосредственной форме. Каждая заключенная в круглые скобки инструкция Лиспа является вызовом функции Лиспа. Первое обозначение внутри скобок является именем функции, а остальные лексемы, разделенные пробелами, являются аргументами, предназначенными для этой функции.

Присваивание значения переменной

Наиболее распространенная функция в файле .emacs — setq, занимается простым присваиванием значения глобальной переменной. В качестве первого аргумента используется имя переменной, которой присваивается значение, а в качестве второго — присваиваемое значение. В следующем примере переменой по имени `c-indent-level` присваивается значение 8:

```
(setq c-indent-level 8)
```

При работе с emacs команда CONTROL+H v выводит приглашение на ввод имени переменной. Если ввести имя переменной и нажать ВВОД, emacs выводит значение переменной.

Установка для переменной значения по умолчанию

Можно установить закрытое в пределах буфера значение переменной по умолчанию, воспользовавшись функцией по имени setq-default. Чтобы установить значение для конкретного элемента вектора, используется функция по имени aset. В качестве ее первого аргумента используется имя вектора, в качестве второго — смещение, а в качестве третьего — значение целевой записи. В файле запуска новыми значениями обычно бывают константы. В табл. 7.21 показаны форматы этих констант.

Вывод значения переменной

Таблица 7.21. Форматы констант, использующихся в файле .emacs

Команда	Результат
Числа	Десятичные целые числа, дополняемые по необходимости знаком минус
Строки	Подобна строке в языке C, но с расширениями для CONTROL- и META-символов: из `\C-s` получается CONTROL+S, из `\M+s` получается META+s, из `\M-\C-s` получается CONTROL+META+s
Символы	В отличие от символов в языке C начинаются со знака ?, за которым следует печатный символ или отключенная обратным слэшем последовательность (например, `?a`, `?\C-i`, `?\033`)
Булевы значения	Вместо 1 и 0 используется `t` для истины (true) и `nil` для лжи (false)
Другие Лисп-объекты	Начинаются с одинарной кавычки, за которой следует имя объекта

Изменение привязки клавиш

Каждый проход командного цикла emacs начинается с перевода поступивших кодов нажатых клавиш в имена исполняемых команд. В основной операции перевода используется ASCII-значение поступившего символа, используемое в качестве индекса для 128-элементного вектора, называемого *картой привязки*.

Иногда восьмой бит кода символа интерпретируется как использование META-*регистра*, но на это можно полагаться не во всех случаях. С точки зрения перевода все META-символы появляются с префиксом ESCAPE, независимо от того, были они или не были набраны именно таким образом.

Каждая позиция этого вектора может иметь одно из следующих содержимых:

- Не определено: в этой карте перевод невозможен.
- Имя другой карты привязки: вызывает переключение на другую карту привязки и ждет поступления кода следующего символа.
- Имя вызываемой функции Лиспа: процесс перевода завершен; вызывается эта функция, выполняющая команду.

Поскольку карты привязки могут ссылаться на другие карты привязки, можно выстроить деревья распознавания произвольной сложности. В основных привязках, применяемых в emacs, используются максимум три клавиши, при этом используется весьма ограниченная группа хорошо известных *префиксных клавиш*, каждая из которых имеет общеизвестное имя карты привязки.

У каждого буфера может быть *локальная карта привязки*, которая в первую очередь используется для любых кодов клавиатуры, поступающих в тот момент, когда выбрано окно, в котором отображается данный буфер. Локальная карта привязки позволяет расширить или переписать обычную привязку под использование в конкретном буфере и чаще всего используется для добавления привязок для основного режима.

Основной поток перевода работает следующим образом:

- Код первого символа проецируется на локальную карту привязки буфера. Если он проецируется на имя функции Лиспа, перевод завершается, и emacs выполняет эту функцию. Если определений для этого кода нет, этот же самый код используется в качестве индекса для глобальной карты привязки верхнего уровня.
- Код символа проецируется на глобальную карту привязки верхнего уровня `global-map`.

Теперь на этой и последующих стадиях выполняются следующие условия:

- Если запись для этого кода символа не определена, возникает ошибка. Терминалу посылается сигнал «звонок», и происходит сброс всех символов, набранных в этой клавишной последовательности.
- Если запись для этого кода символа определена как имя функции Лиспа, перевод завершается, и выполняется эта функция.
- Если запись для этого кода символа определена как имя другой карты привязки, происходит переключение на эту карту привязки и ожидается поступление другого кода символа для выбора одного из этих элементов.

Любой ввод в процессе изменения привязки должен быть либо командой, либо ошибкой. Обычные символы, которые должны быть вставлены в буфер, привязаны, как правило, к команде `self-insert-command`. Каждый из известных префиксных кодов символов связан со своей картой привязки (табл. 7.22).

Таблица 7.22. Префиксы карт привязки

Префикс карты привязки	Применяется для
ctl-x-map	символов, следующих за CONTROL+X
ctl-x-4-map	символов, следующих за CONTROL+X 4
esc-map	символов, следующих за ESCAPE (включая META-символы)
help-map	символов, следующих за CONTROL+H
mode-specific-map	символов, следующих за CONTROL+C

Чтобы узнать о текущем состоянии карт привязки, следует ввести команду CONTROL+H b. Они появятся в следующем порядке: локальные, глобальные и укороченные карты для каждой префиксной клавиши. В каждой строке указывается имя вызываемой функции. Документация по этой функции может быть получена с помощью команды CONTROL+H f *имя_функции* или CONTROL+H k *клавишная_последовательность*.

Чаще всего пользовательская настройка карт привязки сводится к внесению незначительных изменений в привязку глобальных команд без создания каких-нибудь новых карт привязки или команд. Такую настройку проще всего провести в файле .emacs, используя функцию Лиспа `define-key`. Эта функция воспринимает три аргумента:

- Имя карты привязки.
- Одиночный символ, определяющий позицию в этой карте.
- Команду, выполняемую при появлении кода этого символа.

Например, чтобы привязать команду `backward-kill-word` к клавише CONTROL+W, используется инструкция

```
(define-key global-map "\C-w" 'backward-kill-word)
```

Символ \ заставляет интерпретировать сочетание `C-w` как CONTROL+W, а не как три буквы (является эквивалентом \^w). Непарная одинарная кавычка в начале имени команды является нормой. Этот символ в Лиспе предохраняет имя от преждевременного вычисления функции. Для привязки команды `kill-region` к клавишной последовательности CONTROL+X CONTROL+K используется следующая инструкция

```
(define-key ctl-x-map "\C-k" 'kill-region)
```

Образец файла .emacs

Следующий файл ~/.emacs создает обычную среду редактирования, максимально привычную для пользователей редакторов vi и vim. Если какая-нибудь секция или строка вам не подходит,

ее можно отредактировать или превратить в комментарий путем помещения в первой позиции одной или нескольких точек с запятой (;).

```
;;; Preference Variables
(setq make-backup-files nil)          ;Do not make backup files
(setq backup-by-copying t)            ;If you do, at least do not destroy links
(setq delete-auto-save-files t)       ;Delete autosave files when writing orig
(setq blink-matching-paren nil)       ;Do not blink opening delim
(setq require-final-newline 'ask)     ;Ask about missing final newline
;; Reverse mappings for C-h and DEL.
;; Sometimes useful to get DEL character from the Backspace key,
;; and online help from the Delete key.
;; NB: F1 is always bound to online help.
(keyboard-translate ?\C-h ?\177)
(keyboard-translate ?\177 ?\C-h)
;; Some vi sugar: emulate the CR command
;; that positions us to first non-blank on next line.
(defun forward-line-1-skipws ()
  "Position to first nonwhitespace character on next line.»
  (interactive)
  (if (= (forward-line) 0)            ;if moved OK to start of next line
      (skip-chars-forward " \t")))    ;skip over horizontal whitespace
;; Bind this to M-n. ("enhanced next-line")
;; C-M-n is arguably more "correct" but (1) it takes three fingers
;; and (2) C-M-n is already bound to forward-list.
(define-key esc-map "n" 'forward-line-1-skipws)
;; C mode customization: set vanilla (8-space bsd) indention style
(require 'cc-mode)                    ;kiss: be sure it's here
(setq c-default-style
  '(
    (java-mode . "java")
    (awk-mode . "awk")
    (c-mode . "bsd")
    (other . "gnu")
   ))
;; See also CC Mode in online help for more style setup examples.
;; end of c mode style setup
```

Дополнительная информация

По редактору emacs доступен большой объем информации как в электронном, так и бумажном виде. Вполне достаточная информация предоставлена в info-странице по emacs и в функциях справки (см. раздел «Интерактивная справка»). Можно также обратиться к веб-странице, посвященной GNU emacs по адресу www.gnu.org/software/emacs. Поддержка emacs и главные обсуждения, касающиеся работы с этим редактором, представлены в группах новостей comp.emacs и gnu.emacs.help.

Доступ к emacs

Редактор emacs включен в хранилища большинства дистрибутивов Linux. Загрузить и установить emacs можно вместе с коллекцией утилит apt-get, предназначенной для загрузки, установки, удаления, обновления пакетов программного обеспечения

и получения сообщения о них, или вместе с утилитой yum. Самую последнюю версию исходного кода можно загрузить с веб-сайта www.gnu.org.

Связаться с Фондом свободного программного обеспечения можно по следующим адресам:

Почта:

Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301, USA

Электронная почта: gnu@gnu.org
Телефон: +1 617-542-5942
Факс: +1 617 542 2652
Интернет: www.gnu.org

Заключение

Перед многими командами, представленными в следующих таблицах, можно ставить числовые аргументы, чтобы повторять команду заданное количество раз. Чтобы предотвратить ввод в emacs аргументов в виде текста, перед вводом числового аргумента следует ввести команду CONTROL+U.

В табл. 7.23 перечислены команды перемещения курсора.

Таблица 7.23. Перемещение курсора

Команда	Результат
CONTROL+F	Посимвольное перемещение вперед
CONTROL+B	Посимвольное перемещение назад
META+f	Пословное перемещение вперед
META+b	Пословное перемещение назад
META+e	Перемещение к концу предложения
META+a	Перемещение к началу предложения
META+}	Перемещение к концу абзаца
META+{	Перемещение к началу абзаца
META+>	Перемещение к концу буфера
META+<	Перемещение к началу буфера
CONTROL+ESCAPE	Перемещение к концу строки
CONTROL+A	Перемещение к началу строки
CONTROL+N	Перемещение вперед (вниз) на одну строку
CONTROL+P	Перемещение назад (вверх) на одну строку
CONTROL+V	Прокрутка вперед (вниз) на одно окно
META+v	Прокрутка назад (вверх) на одно окно
CONTROL+L	Очистка и перерисовка окна и прокрутка текущей строки на середину окна
META+r	Перемещение к началу средней строки
CONTROL+U *num* META+r	Перемещение к строке с номером *num* (0 = к самой верхней строке, – = к самой нижней строке)

В табл. 7.24 перечислены команды, стирающие и удаляющие текст.

Таблица 7.24. Стирание и удаление текста

Команда	Результат
CONTROL+DELETE	Удаление символа под курсором
DELETE	Удаление символа слева от курсора
META+d	Стирание вперед до конца текущего слова
META+DELETE	Стирание назад до начала предыдущего слова
META+k	Стирание вперед до конца предложения
CONTROL+X DELETE	Стирание назад до начала предложения
CONTROL+K	Стирание вперед вплоть до конца строки, но исключительно символа перевода строки, которым она завершается; если между курсором и символом перевода строки текст отсутствует, то стирание символа перевода строки
CONTROL+U 1 CONTROL+K	Стирание от курсора вперед, включая и символ перевода строки
CONTROL+U 0 CONTROL+K	Стирание от курсора назад к началу строки
META+z *char*	Стирание вперед до следующего появления *char*, но исключая его
META+w	Копирование области в кольцевой буфер стирания (без удаления области из буфера)
CONTROL+W	Стирание области (с удалением ее из буфера)
CONTROL+Y	Восстановление самого последнего стертого текста в текущем буфере на месте указателя; установка метки на начало этого текста, а указателя и курсора — на его конец
META+y	Удаление только что вставленного текста, прокрутка кольцевого буфера стирания и восстановление его следующей записи (только после CONTROL+Y или META+y)

В табл. 7.25 перечислены команды, выполняющие поиск строк и соответствий регулярным выражениям.

Таблица 7.25. Команды поиска

Команда	Результат
CONTROL+S	Приглашение на ввод строки для поиска вперед с последовательным уточнением
CONTROL+S ВВОД	Приглашение на ввод всей строки для поиска вперед
CONTROL+R	Приглашение на ввод строки для поиска назад с последовательным уточнением
CONTROL+R ВВОД	Приглашение на ввод всей строки для поиска назад
META+CONTROL+S	Приглашение на ввод регулярного выражения для поиска вперед соответствующего ему текста с последовательным уточнением
META+ CONTROL+S ВВОД	Приглашение на ввод всего регулярного выражения для поиска вперед соответствующего ему текста
META+CONTROL+R	Приглашение на ввод регулярного выражения для поиска назад соответствующего ему текста с последовательным уточнением
META+ CONTROL+R ВВОД	Приглашение на ввод всего регулярного выражения для поиска назад соответствующего ему текста

В табл. 7.26 перечислены команды, предоставляющие интерактивную справку.

Таблица 7.26. Интерактивная справка

Команда	Результат
CONTROL+H a	Приглашение на ввод строки и вывод списка команд, в чьем имени есть эта строка
CONTROL+H b	Вывод длинной таблицы всех действующих на данный момент клавиш, привязанных к командам
CONTROL+H c *клавишная_последовательность*	Вывод имени команды, привязанной к *клавишной_последовательности*
CONTROL+H k *клавишная_последовательность*	Вывод имени команды, привязанной к *клавишной_последовательности* и соответствующей документации
CONTROL+H f	Приглашение на ввод имени Лисп-функции и вывод документации по этой функции
CONTROL+H i (буква «i» в нижнем регистре)	Вывод главного меню утилиты info (см. раздел « info: Отображение информации об утилитах» на с. 56)
CONTROL+H l (буква «l» в нижнем регистре)	Вывод последних 100 набранных символов
CONTROL+H m	Вывод документации и специальных клавишных привязок для текущего основного режима
CONTROL+H n	Вывод файла новостей emacs
CONTROL+H t	Запуск сеанса работы с учебным пособием по emacs
CONTROL+H v	Приглашение на ввод имени Лисп-переменной и вывод документации по этой переменной
CONTROL+H w	Приглашение на ввод имени команды и вывод клавишной последовательности, привязанной к этой команде, если таковая имеется

В табл. 7.27 перечислены команды, работающие с областью.

Таблица 7.27. Работа с областью

Команда	Результат
META+W	Копирование области в кольцевой буфер стирания без ее удаления
CONTROL+W	Стирание (удаление) области
META+x `print-region`	Копирование области в спулер принтера
META+x `append-to-buffer`	Приглашение на ввод имени буфера и добавление области к этому буферу
META+x `append-to-file`	Приглашение на ввод имени файла и добавление области к этому файлу
CONTROL+X CONTROL+U	Перевод букв области в верхний регистр
CONTROL+X CONTROL+L	Перевод букв области в нижний регистр

В табл. 7.28 перечислены команды, работающие со строками.

Таблица 7.28. Работа со строками

Команда	Результат
META+x occur	Приглашение на ввод регулярного выражения и внесение каждой строки, в которой найдено соответствие выражению, в буфер по имени *Occur*
META+x delete-matching-lines	Приглашение на ввод регулярного выражения и удаление вперед от указателя тех строк, в которых найдено соответствие регулярному выражению
META+x delete-non-matching-lines	Приглашение на ввод регулярного выражения и удаление вперед от указателя тех строк, в которых не найдено соответствие регулярному выражению

В табл. 7.29 перечислены команды, заменяющие строки и текст, соответствующий регулярным выражениям в безусловном и интерактивном режиме.

Таблица 7.29. Команды, заменяющие текст

Команда	Результат
META+x replace-string	Приглашение на ввод двух строк и замена каждого экземпляра первой строки второй строкой от указателя вперед; установка метки на то место, где вводилась команда
META+% или META+x query-replace	Результат такой же, как и при вводе предыдущей команды, но с запросом на каждую замену (список ответов см. в табл. 7.30)
META+x replace-regexp	Приглашение на ввод регулярного выражения и строки и замена каждого найденного текста, соответствующего регулярному выражению, строкой; установка метки на то место, где вводилась команда
META+x query-replace-regexp	Результат такой же, как и при вводе предыдущей команды, но с запросом на каждую замену (список ответов см. в табл. 7.30)

В табл. 7.30 перечислены ответы на запросы на замену.

Таблица 7.30. Ответы на запросы на замену

Команда	Результат
ВВОД	Прекращение поиска (прекращение замен или запросов на дальнейшие замены)
ПРОБЕЛ	Осуществление замены и продолжение выдачи запросов на замену
DELETE	Отказ от текущей замены и продолжение выдачи запросов на замену
, (запятая)	Осуществление замены, вывод ее результата и запрос следующей команды
. (точка)	Осуществление текущей замены и прекращение замен или запросов на их проведение
! (восклицательный знак)	Замена текущего и всех последующих экземпляров без выдачи запросов

В табл. 7.31 перечислены команды для работы с окнами.

Таблица 7.31. Работа с окнами

Команда	Результат
CONTROL+X b	Приглашение на ввод имени другого буфера и отображение содержимого этого буфера в текущем окне
CONTROL+X 2	Разбиение текущего окна на две части по вертикали
CONTROL+X 3	Разбиение текущего окна на две части по горизонтали
CONTROL+X o (буква «о» в нижнем регистре)	Выбор другого окна
META+CONTROL+V	Прокрутка другого окна
CONTROL+X 4b	Приглашение на ввод имени буфера и выбор его в другом окне
CONTROL+X 4f	Приглашение на ввод имени файла и выбор его в другом окне
CONTROL+X 0 (нуль)	Удаление текущего окна
CONTROL+X 1 (единица)	Удаление всех окон, кроме текущего
META+x `shrink-window`	Уменьшение текущего окна на одну строку
CONTROL+X ^	Удлинение текущего окна на одну строку
CONTROL+X }	Увеличение ширины текущего окна на один символ
CONTROL+X {	Уменьшение ширины текущего окна на один символ

В табл. 7.32 перечислены команды для работы с файлами.

Таблица 7.32. Работа с файлами

Команда	Результат
CONTROL+X CONTROL+F	Приглашение на ввод имени файла и считывание его содержимого в новый буфер; присваивание буферу в качестве имени упрощенного имени файла
CONTROL+X CONTROL+V	Приглашение на ввод имени файла и считывание его содержимого в текущий буфер (с перезаписью содержимого текущего буфера)
CONTROL+X 4 CONTROL+F	Приглашение на ввод имени файла и считывание его содержимого в новый буфер; присваивание буферу в качестве имени упрощенного имени файла. Создание нового окна для этого буфера и выбор этого окна. Если на экране до этого было только одно окно, то эта команда разбивает экран пополам
CONTROL+X CONTROL+S	Сохранение текущего буфера в исходном файле
CONTROL+X s	Приглашение на сохранение каждого измененного буфера (y/n)
META+x `set-visited-file-name`	Приглашение на ввод имени файла и установка его имени как «исходного» имени текущего буфера
CONTROL+X CONTROL+W	Приглашение на ввод имени файла, установка этого имени как «исходного» имени для текущего буфера и сохранение текущего буфера в этом файле
META+~ (тильда)	Сброс флажка модификации текущего буфера. При использовании следует проявлять осмотрительность

В табл. 7.33 перечислены команды для работы с буферами.

Таблица 7.33. Работа с буферами

Команда	Результат
CONTROL+X CONTROL+S	Сохранение текущего буфера в связанном с ним файле
CONTROL+X CONTROL+F	Приглашение на ввод имени файла и посещение (открытые) этого файла
CONTROL+X b	Приглашение на ввод имени буфера и выбор этого буфера. Если буфер отсутствует, создание буфер
CONTROL+X 4b	Приглашение на ввод имени буфера и выбор его в другом окне. Существующее окно не затрагивается, хотя новое окно может его перекрыть
CONTROL+X CONTROL+B	Создание буфера по имени *Buffer List*, и вывод его в другом окне. Существующее окно не затрагивается, хотя новое окно может его перекрыть. Новый буфер не выбирается. В буфере *Buffer List* отображаются данные каждого буфера, включая имя, размер, режим (режимы) и имя исходного файла
META+x rename-buffer	Приглашение на ввод нового имени буфера и присваивание этого имени текущему буфер
CONTROL+X CONTROL+Q	Переключение текущего буфера в статус и из статуса только для чтения и вывод соответствующего индикатора %% в Строке режима
META+x append-to-buffer	Приглашение на ввод имени буфера и добавление области к концу этого буфера
META+x prepend-to-buffer	Приглашение на ввод имени буфера и добавление области в начало этого буфера
META+x copy-to-buffer	Приглашение на ввод имени буфера, удаление содержимого этого буфера и копирование области в этот буфер
META+x insert-buffer	Приглашение на ввод имени буфера и вставка всего содержимого этого буфера в текущий буфер в текущую позицию указателя
CONTROL+X k	Приглашение на ввод имени буфера и удаление этого буфера
META+x kill-some-buffers	Перебор всего списка буферов и предоставление возможности удаления каждого буфера

В табл. 7.34 перечислены команды, запускающие команды оболочки в активном режиме. В некоторых оболочках эти команды могут не работать.

Таблица 7.34. Команды, работающие с командами оболочки в активном режиме

Команда	Результат
META+! (восклицательный знак)	Приглашение на ввод команды оболочки, выполнение этой команды и отображение выходной информации
CONTROL+U META+! (восклицательный знак)	Приглашение на ввод команды оболочки, выполнение этой команды и вставка выходной информации в место, где находится указатель
META+| (вертикальная черта)	Приглашение на ввод команды оболочки, предоставление ей области в качестве ввода, фильтрация ее через команду и отображение выходной информации команды

Команда	Результат
CONTROL+U META+| (вертикальная черта)	Приглашение на ввод команды оболочки, предоставление ей области в качестве ввода, фильтрация ее через команду, удаление старой области и вставка выходной информации команды вместо области

В табл. 7.35 перечислены команды, запускающие команды оболочки в фоновом режиме.

Таблица 7.35. Команды, работающие с командами оболочки в фоновом режиме

Команда	Результат
META+x `compile`	Приглашение на ввод команды оболочки и запуск этой команды в фоновом режиме с направлением выходной информации в буфер по имени *`compilation`*
META+x `kill-compilation`	Прекращение фонового процесса

В табл. 7.35 перечислены команды, преобразующие буквы текста из верхнего в нижний регистр и обратно.

Таблица 7.36. Команды преобразования регистра букв

Команда	Результат
META+l (буква «l» в нижнем регистре)	Перевод символов слова, находящегося справа от указателя, в нижний регистр
META+u	Перевод символов слова, находящегося справа от указателя, в верхний регистр
META+c	Перевод символов слова, находящегося справа от указателя, в регистр с первыми заглавными буквами
CONTROL+X CONTROL+L	Перевод символов области в нижний регистр
CONTROL+X CONTROL+U	Перевод символов области в верхний регистр

В табл. 7.37 перечислены команды для работы в режиме языка C.

Таблица 7.37. Команды режима языка C

Команда	Результат
CONTROL+META+f	Перемещение вперед на одно выражение
CONTROL+META+b	Перемещение назад на одно выражение
CONTROL+META+k	Перемещение вперед на одно выражение и удаление этого выражения
CONTROL+META+@	Установка метки на ту позицию, куда она была бы перемещена командой CONTROL+META+f, без изменения позиции указателя
CONTROL+META+a	Перемещение к началу самого последнего определения функции
CONTROL+META+e	Перемещение к концу следующего определения функции
CONTROL+META+h	Перемещение указателя к началу, а метки к концу текущего определения функции (или следующего определения, если указатель находится между определениями)

Для создания буфера по имени *shell* и запуска подчиненной оболочки следует ввести команду META+x shell. В табл. 7.38 перечислены команды, работающие с этим буфером.

Таблица 7.38. Команды для работы в режиме оболочки

Команда	Результат
ВВОД	Отправка текущей строки оболочке
CONTROL+C CONTROL+D	Отправка символа CONTROL+D оболочке или ее подчиненной оболочке
CONTROL+C CONTROL+C	Отправка символа CONTROL+C оболочке или ее подчиненной оболочке
CONTROL+C CONTROL+\	Отправка сигнала quit оболочке или ее подчиненной оболочке
CONTROL+C CONTROL+U	Стирание текста в текущей незавершенной строке
CONTROL+C CONTROL+R	Прокрутка назад к началу последней, выведенной оболочкой информации, помещение первой строки выведенной информации в верхнюю часть окна
CONTROL+C CONTROL+O (буква «O» в верхнем регистре)	Удаление последнего пакета информации, выведенной оболочкой

Упражнения

1. Заданный буфер a заполнен английским текстом, ответьте на следующие вопросы:
   1) Как заменить все экземпляры слова his словом hers?
   2) Как осуществить эту замену только в последнем абзаце?
   3) Существует ли способ просмотреть каждое применение этого слова в контексте перед его заменой?
   4) Как справиться с возможным случаем присутствия слова His в начале предложения?
2. Какая команда вызывает перемещение курсора к концу текущего абзаца? Можно ли использовать эту команду для перемещения по буферу от абзаца к абзацу?
3. Предположим, что вы потеряли мысль в середине набора длинного предложения.
   1) Существует ли простой способ, позволяющий стереть неудачное предложение и начать все заново?
   2) Что делать, если неправильным оказалось только одно слово? Существует ли альтернативный способ его побуквенного удаления?
4. Когда на работу с абзацем уже было потрачено некоторое время, то вполне вероятно, что некоторые строки получились слишком короткими, а некоторые — слишком длинными. Существует ли команда для приведения абзаца в порядок без повторной разбивки всех строк вручную?
5. Существует ли способ изменения всего содержимого буфера, превращающий все буквы в заглавные? Можно ли придумать способ изменения только одного абзаца?
6. Как поменять местами два абзаца?
7. Как поменять местами два слова?
8. Представьте, что вы увидели сообщение телеконференции, содержащее что-то забавное, и сохранили это сообщение в файле. Как вставить этот файл в ваш собственный буфер? Что

делать, если вам захочется воспользоваться только двумя абзацами из этого сообщения? Как можно будет добавить символ > к началу каждой включаемой строки?

9. При использовании одной только клавиатуры emacs всегда предлагает полный набор возможностей редактирования. Как правило, одни и те же цели для решения любой задачи редактирования могут быть достигнуты целым рядом технологических приемов. В среде X выбор еще больше расширяется благодаря добавлению новой группы визуальных альтернатив, предусматривающих применение мыши. Исходя из этих возможностей вы должны выбрать подходящий вам способ для наилучшего решения следующей задачи редактирования.

   Рассмотрите следующий фрагмент творчества Шекспира:

```
 1. Full fathom five thy father lies;
 2. Of his bones are coral made;
 3. Those are pearls that were his eyes:
 4. Nothing of him that doth fade,
 5. But doth suffer a sea-change
 6. Into something rich and strange.
 7. Sea-nymphs hourly ring his knell:
 8. Ding-dong.
 9. Hark! now I hear them--
10. Ding-dong, bell!
```

   Следующий фрагмент был набран с ошибками:

```
 1. Full fathiom five tyy father lies;
 2. These are pearls that were his eyes:
 3. Of his bones are coral made;
 4. Nothin of him that doth fade,
 5. But doth susffer a sea-change
 6. Into something rich and strange.
 7. Sea-nymphs hourly ring his knell:
 8. Ding=dong.
 9. Hard! now I hear them--
10. Ding-dong, bell!
```

   Для ответа на следующие вопросы нужно использовать только клавиатуру:

   1) Сколько способов вы можете придумать для перемещения курсора на орфографические ошибки?
   2) Сколько способов исправления ошибок можно придумать, когда курсор находится на ошибках или возле них?
   3) Существуют ли способы исправления ошибок без явного перемещения на них или без их поиска? Сколько таких способов вы можете придумать?
   4) Строки 2 и 3 в перенабранном материале переставлены местами. Сколько способов вы можете придумать для исправления данной ситуации?

Упражнение повышенной сложности

Предположим, что ваш буфер содержит показанный здесь код на языке C, Основной режим установлен на использование языка C, и курсор установлен на конец строки `while`, что показано черным прямоугольником:

```
/*
 * Copy string s2 to s1.  s1 must be large enough
 * return s1
 */
char *strcpy(char *s1, char *s2)
```

```
{
        char *os1;
        os1 = s1;
        while (*s1++ = *s2++)
                ;
        return os1;
}
/*
 * Copy source into dest, stopping after '\0' is copied, and
 * return a pointer to the '\0' at the end of dest. Then our caller
 * can catenate to the dest  * string without another strlen call.
 */
char *stpcpy (char *dest, char *source)
{
        while ((*dest++ = *source++) != '\0') ■
                ; /* void loop body */
        return (dest - 1);
}
```

1) Какая команда перемещает курсор к открывающей фигурной скобке процедуры `strcpy`? Какая команда перемещает курсор за закрывающую фигурную скобку? Можно ли использовать эти команды для прохода по буферу от процедуры к процедуре?
2) Предположим, курсор только что прошел закрывающие круглые скобки условия `while`. Как переместить его к составляющей пару открывающей круглой скобке? Как вернуть курсор назад на составляющую пару закрывающую круглую скобку? Работают ли те же команды для пар `[ ]` (квадратных скобок) и `{ }` (фигурных скобок)? Чем отличается их работа от vim-команды %?
3) Одна из процедур имеет отступ в стиле Berkeley, а другая — в стиле GNU. Какая команда изменит отступ строки в соответствии с текущим, установленным вами стилем отступов? Как изменить отступы во всей процедуре?
4) Предположим, что вы захотели написать процедуру из пяти строк и собираетесь использовать процедуру `strcpy` в качестве исходного материала для дальнейшего редактирования. Как можно сделать пять копий процедуры `strcpy`?
5) Как откомпилировать код без выхода из emacs?

ЧАСТЬ III

Оболочки

Глава 8. Оболочка Bourne Again Shell 282

Глава 9. Оболочка TC Shell 367

Глава 8. Оболочка Bourne Again Shell

Эта глава является продолжением главы 5 и посвящена оболочке Bourne Again Shell (bash). В ней особо отмечаются те моменты, где у оболочки tcsh реализация функций отличается от bash, и, если это уместно, дается ссылка на главу и раздел, где рассматривается альтернативная реализация. Глава 10 является логическим продолжением данной главы, в ней рассматриваются команды управления процессом выполнения сценариев и более сложные аспекты программирования оболочки Bourne Again Shell (bash). Главная страница bash находится по адресу www.gnu.org/software/bash. Info-страница bash содержит полноценное справочное руководство по Bourne Again Shell.

Оболочки Bourne Again Shell и TC Shell (tcsh) являются командными интерпретаторами и языками высокого уровня. В качестве командных интерпретаторов они обрабатывают те команды, которые вводятся в командную строку в ответ на приглашение. Когда оболочка используется как язык программирования, она обрабатывает команды, сохраненные в файлах, называемых *сценариями оболочки*. Как и у других языков программирования, у оболочки есть переменные и команды управления (например, циклы `for` и инструкции `if`).

Когда оболочка используется в качестве командного интерпретатора, вы можете настроить свое собственное рабочее окружение. Можно сделать собственную форму приглашения, отображающую имя рабочего каталога, создать функцию или псевдоним для утилиты cp, который не давал бы ей переписывать файлы определенного типа, можно воспользоваться преимуществами переменных, являющихся ключевыми словами, для изменения аспектов работы оболочки и т. д. Можно также писать сценарии оболочки, которые доводят до нее ваши команды, — все, что угодно из однострочного сценария, в котором хранится длинная, сложная команда, запускающая набор отчетов, осуществляющая их вывод и отправляющая вам по почте сообщение, когда задание будет выполнено. Более сложные сценарии оболочки являются настоящими программами, которые не просто запускают другие программы. В главе 10 приводятся некоторые примеры сценариев такого рода.

Большинство системных сценариев оболочки пишутся для запуска под управлением bash (или dash, см. ниже). Если вам придется когда-нибудь работать в однопользовательском или в восстановительном режиме — когда вы, к примеру, запускаете систему или осуществляете ее обслуживание, администрирование или восстановление, то лучше познакомиться с этой оболочкой поближе.

В этой главе приводится расширенное рассмотрение интерактивных свойств оболочки, описание которых давалось в главе 5, при этом объясняется, как создавать и запускать простые сценарии оболочки, рассматривается управление заданиями. Также читатель знакомится с основными аспектами программирования оболочки, рассматриваются история и псевдонимы, описывается расширение командной строки. В главе 9 дается описание использования в интерактивном режиме оболочки TC Shell, а также рассматривается программирование этой оболочки, а в главе 10 рассматриваются более сложные проблемы программирования оболочек.

Предыстория

В основе оболочки Bourne Again Shell лежит оболочка Bourne Shell, одна из первых оболочек UNIX (чтобы избежать недоразумений, в данной книге на нее дается ссылка как на исходную оболочку Bourne Shell), созданная Стивом Борном (Steve Bourne), сотрудником лаборатории Bell Laboratories компании AT&T. С годами исходная оболочка Bourne Shell была расширена, но при этом оставалась основной оболочкой, поставляемой со многими коммерческими версиями UNIX.

Оболочка sh

Благодаря своей долгой и успешной истории оригинальная оболочка Bourne Shell использовалась для написания множества сценариев оболочки, помогающих управлять UNIX-системами. Некоторые из этих сценариев появились в Linux как сценарии оболочки Bourne Again Shell. Хотя оболочка Bourne Again Shell включает в себя множество расширений и свойств, которых нет в исходной оболочке Bourne Shell, bash поддерживает совместимость с исходной оболочкой Bourne Shell, поэтому сценарии этой оболочки можно запускать под управлением bash. На UNIX-системах исходную оболочку Bourne Shell называют sh.

На некоторых Linux-системах sh является символьной ссылкой на bash, обеспечивая запуск тех сценариев, для которых требуется присутствие оболочки Bourne Shell. При вызове по ссылке sh bash делает все возможное для эмуляции исходной оболочки Bourne Shell.

Оболочка dash

Исполняемый файл bash занимает около 800 Кбайт, содержит множество функций и хорошо работает в качестве оболочки для входа пользователя в систему. Оболочка dash (Debian Almquist) занимает около 100 Кбайт, обеспечивает совместимость с оболочкой Bourne Shell, позволяющую выполнять сценарии оболочки (использовать ее в неинтерактивном режиме), и благодаря своему размеру может загружать и выполнять сценарии значительно быстрее, чем bash. Большинство системных сценариев настроено под запуск sh, что при работе под управлением некоторых систем, таких как Ubuntu, является символьной ссылкой на dash. Такая настройка позволяет системе запуститься и быстро запустить системные сценарии оболочки.

Оболочка Korn Shell

В System V UNIX появилась оболочка Korn Shell (ksh), написанная Дэвидом Корном (David Korn). В этой оболочке представлены расширенные версии многих свойств, имеющихся в исходной оболочке Bourne Shell, и добавлено множество новых свойств. Некоторые свойства оболочки Bourne Again Shell, например псевдонимы команды и редактирование командных строк, основаны на похожих функциях из оболочки Korn Shell.

POSIX

POSIX (Portable Operating System Interface — интерфейс переносимой операционной системы) — это семейство взаимосвязанных стандартов, разработанных комитетом по стандартам переносимых приложений — PASC (работающим в рамках IEEE Portable Application Standards Committee, www.pasc.org). Мощная система ответов на наиболее часто задаваемые вопросы (FAQ) по POSIX, включающая множество ссылок, находится по адресу www.opengroup.org/austin/papers/posix_faq.html.

В POSIX-стандарте 1003.2 детально описаны функциональные требования к оболочкам. Оболочка Bourne Again Shell обладает свойствами, которые соответствуют

требованиям этого стандарта. Прилагаются немалые усилия для того, чтобы сделать оболочку Bourne Again Shell полностью отвечающей стандарту POSIX. Если вызвать bash с ключом `--posix`, то поведение оболочки Bourne Again Shell будет практически соответствовать требованиям стандарта POSIX.

INFO-ДОКУМЕНТАЦИЯ ПО ОБОЛОЧКЕ BOURNE AGAIN SHELL СОДЕРЖИТСЯ В ПАКЕТЕ BASH-DOC

Man-страница по bash доступна и без установки пакета bash-doc. Установка этого пакета предоставляет более полные info-страницы по bash. Дополнительные сведения по info даны в разделе «info: Отображение информации об утилитах» на с. 56.

Основы оболочки

В этом разделе рассматривается написание и использование файлов запуска, перенаправление стандартных ошибок, написание и выполнение простых сценариев оболочки, разделение и группировка команд, реализация управления заданиями и работа со стеком каталогов.

CHSH: СМЕНИТЕ ОБОЛОЧКУ ВХОДА В СИСТЕМУ

Специалист, настраивающий вашу учетную запись, определяет, какой оболочкой вы пользуетесь при первом входе в систему или при открытии окна эмулятора терминала в среде GUI. Под управлением большинства дистрибутивов Linux bash является оболочкой по умолчанию. После входа в систему вы можете запустить любую, понравившуюся вам оболочку. Введите имя оболочки, которую нужно использовать (bash, tcsh или имя другой оболочки), и нажмите ВВОД. Следующее приглашение на ввод команды будет выведено уже новой оболочкой. Чтобы вернуться в предыдущую оболочку, нужно ввести команду exit. Поскольку запускаемые таким образом оболочки являются вложенными друг в друга (одна запускается поверх другой), вы сможете выйти из системы только из исходной оболочки. Когда используются несколько вложенных оболочек, нужно вводить команду exit до тех пор, пока вы не окажетесь в своей исходной оболочке. Только тогда вы сможете выйти из системы.

Для полной смены оболочки, из которой вы вошли в систему, следует воспользоваться утилитой chsh. Сначала нужно ввести команду chsh. В ответ на приглашение следует ввести свой пароль и абсолютное имя той оболочки, которую вы хотите использовать (/bin/bash, /bin/tcsh или имя какой-нибудь другой оболочки). Когда вы меняете свою оболочку входа в систему, используя эмулятор терминала (см. главу 2, раздел «Вход в систему через терминал или эмулятор терминала»), работающий в GUI, последующие окна эмулятора терминала не будут реагировать на изменения до тех пор, пока вы не выйдете из системы и снова не войдете в нее.

Файлы запуска

При запуске оболочки она самостоятельно запускает файлы для инициализации. Какие именно файлы будут запущены оболочкой, зависит от многих факторов: что это за оболочка, используется ли она для входа в систему, или же это интерактивная оболочка, не использующаяся для входа в систему (например, запускаемая с помощью команды bash), или это неинтерактивная оболочка (используемая для выполнения сценария оболочки). У вас должен быть доступ на чтение файла запуска, чтобы выполнить содержащиеся в нем команды. Обычно дистрибутивы Linux помещают соответствующие команды в некоторые из этих файлов. В данном разделе рассматриваются файлы запуска оболочки bash. Сведения о файлах запуска оболочки tcsh даны в разделе «Файлы запуска» на с. 369.

Оболочки для входа в систему

Файлы, рассматриваемые в данном разделе, выполняются оболочками, предназначенными для входа в систему, и теми оболочками, которые запускаются с помощью ключа bash `--login`. Оболочки для входа в систему по своей природе обладают свойством интерактивности.

Файл /etc/profile

Сначала оболочка выполняет команды, содержащиеся в файле /etc/profile. Пользователь, работающий с правами привилегированного пользователя (root-правами), может настроить этот файл для пользователей, запускающих bash, на установку характеристик по умолчанию, распространяющихся на всю систему.

Файлы .bash_profile, .bash_login и .profile

Затем оболочка ищет файлы ~/.bash_profile, ~/.bash_login и ~/.profile (~/ — сокращение для указания вашего личного каталога) в указанном порядке, выполняя команды, имеющиеся в первом из этих найденных файлов. В один из этих файлов можно поместить команды для переписывания установок по умолчанию, имеющихся в файле /etc/profile. Оболочка, запущенная на виртуальном терминале, не выполняет команды, которые содержатся в этих файлах.

Файл .bash_logout

При выходе из системы bash выполняет команды, которые содержатся в файле ~/.bash_logout. В этом файле часто содержатся команды, которые подчищают все за сеансом работы, например удаляют временные файлы.

Интерактивные оболочки, не предназначенные для входа в систему

Команды в предыдущих файлах запуска не выполняются интерактивными, не предназначенными для входа в систему оболочками. Но эти оболочки наследуют значения из переменных оболочки, предназначенной для входа в систему, которые устанавливаются этими файлами запуска.

Файл /etc/bashrc

Хотя этот файл и не вызывается напрямую оболочкой bash, многие файлы ~/.bashrc вызывают файл /etc/bashrc. Содержащиеся в нем настройки позволяют пользователю работать с root-правами для установки распространяющихся на всю систему характеристик по умолчанию для не предназначенных для входа в систему оболочек bash.

Файл .bashrc

Интерактивная, не предназначенная для входа в систему оболочка выполняет команды, которые находятся в файле ~/.bashrc. Обычно файл запуска для оболочки, предназначенной для входа в систему, такой как .bash_profile, запускает этот файл, поэтому команды, имеющиеся в файле .bashrc, запускаются оболочками, предназначенными и не предназначенными для входа в систему.

Неинтерактивные оболочки

Команды в ранее рассмотренных файлах запуска не выполняются неинтерактивными оболочками, например такими, которые запускают сценарии оболочек. Но эти оболочки наследуют переменные оболочек, предназначенных для входа в систему, значения которых устанавливались файлами запуска.

Переменная BASH_ENV

Неинтерактивные оболочки ищут переменную среды окружения BASH_ENV (или ENV, если оболочка вызвана под именем sh) и выполняют команды, которые находятся в файле, указанном этой переменной.

Настройка файлов запуска

Несмотря на существование множества файлов запуска и типов оболочек, обычно требуются лишь файлы .bash_profile и .bashrc, которые находятся в вашем личном каталоге. Команды, похожие на следующие, предназначенные для файла .bash_profile, запускают команды из файла .bashrc для оболочек, предназначенных для входа в систему (когда файл .bashrc существует). С помощью этих настроек команды, содержащиеся в файле .bashrc, выполняются оболочками как предназначенными, так и не предназначенными для входа в систему.

```
if [ -f ~/.bashrc ]; then . ~/.bashrc; fi
```

Код [-f ~/.bashrc] используется для проверки факта существования файла по имени .bashrc в вашем личном каталоге. Сведения о встроенной утилите test и ее синониме [ ] даны в главе 10 и в части V. А сведения о встроенной утилите . (точка) даны в разделе «Утилита (точка) или source: Запуск файла запуска текущей оболочке» на с. 287.

ДЛЯ УСТАНОВКИ ЗНАЧЕНИЯ ПЕРЕМЕННОЙ PATH, НУЖНО ВОСПОЛЬЗОВАТЬСЯ ФАЙЛОМ .BASH_PROFILE

Так как команды в файле .bashrc могут исполняться многократно, и поскольку подчиненные оболочки наследуют экспортируемые переменные, то команды, добавляющие данные к существующим переменным, лучше использовать в файле .bash_profile. Например, следующая команда добавляет подкаталог bin, находящийся в каталоге home к переменной (см. раздел «Переменные, являющиеся ключевыми словами» на с. 305), и должна находиться в файле .bash_profile:

```
PATH=$PATH:$HOME/bin
```

Когда эта команда помещается в файл .bash_profile, а не в файл .bashrc, строка добавляется к переменной PATH только один раз, при входе в систему. Модификация переменной в файле .bash_profile заставляет изменения, внесенные в интерактивном сеансе, распространяться на подчиненные оболочки. В отличие от этого модификация переменной в файле .bashrc переписывает изменения, унаследованные от родительской оболочки.

Далее приводятся образцы файлов .bash_profile и .bashrc. Некоторые команды, используемые в этих файлах, будут рассмотрены в этой главе чуть позже. В любые файлы запуска нужно переносить те переменные и функции, к которым желателен доступ из дочерних процессов. Дополнительные сведения даны в разделе «Локальность переменных» на с. 454.

```
$ cat ~/.bash_profile
if [ -f ~/.bashrc ]; then
    . ~/.bashrc             # Читает локальный файл запуска,если он существует
fi
PATH=$PATH:.                # Добавляет к PATH рабочий каталог
export PS1='[\h \W \!]\$ '  # Настраивает приглашение
```

Первая команда в предшествующем файле .bash_profile выполняет команды в пользовательском файле .bashrc, если таковой существует. Следующая команда добавляет к переменной PATH (см. раздел «Переменная PATH: Указание тех мест, где оболочка ищет программы» на с. 312). Обычно значение переменной PATH устанавливается и экс-

портируется из файла /etc/profile, поэтому экспортировать ее в пользовательский файл запуска нет необходимости. Заключительная команда устанавливает и экспортирует переменную PS1 (см. раздел «Переменная PS1: Приглашение пользователю (первичное)» на с. 314), управляющую приглашением пользователю на ввод команды.

Далее показан образец файла .bashrc. Первая команда выполняет команды, находящиеся в файле /etc/bashrc, если таковой существует. Следующие занимаются установками для файлов и экспортом переменных LANG (табл. 8.5) и VIMINIT (для инициализации редактора vim) и определяют несколько псевдонимов. Заключительная команда определяет функцию для обмена двух файлов именами.

```
$ cat ~/.bashrc
if [ -f /etc/bashrc ]; then
   source /etc/bashrc    # читает глобальный файл запуска, если тот существует
fi

set -o noclobber           # предотвращает перезапись файлов
unset MAILCHECK            # выключает уведомление о новом сообщении "you have new mail"
export LANG=C              # устанавливает значение переменной LANG
export VIMINIT='set ai aw' # устанавливает выбранные варианты vim
alias df='df -h'           # устанавливает псевдонимы
alias rm='rm -i'           # включает постоянный режим интерактивности для rm
alias lt='ls -ltrh | tail'
alias h='history | tail'
alias ch='chmod 755 '

function switch()          # функция для обмена двух файлов именами
{
   local tmp=$$switch
   mv "$1" $tmp
   mv "$2" "$1"
   mv $tmp "$2"
}
```

Утилита . (точка) или source: Запуск файла запуска в текущей оболочке

Чтобы после редактирования такого файла запуска, как .bashrc, изменения вступили в силу, не нужно выходить из системы и снова в нее входить. Вместо этого нужно запустить файл запуска с помощью встроенной утилиты . (точка) или source (при использовании bash это одна и та же команда, но при использовании оболочки tcsh доступна только команда source, см. табл. 9.10). Как и все остальные команды, команда . (точка) должна сопровождаться в командной строке пробелом. Использование команды . (точка) или source аналогично запуску сценария оболочки, за исключением того, что эти команды запускают оболочку в качестве составной части текущего процесса. Следовательно, когда вы используете . (точка) или source для запуска сценария, те изменения, которые внесены в переменную из этого сценария, воздействуют на оболочку, из которой запущен сценарий. Если файл запуска был запущен как обычный сценарий оболочки, без использования встроенной утилиты . (точка) или source, переменные, которые были созданы в файле запуска, будут действительны только для той подчиненной оболочки, в которой выполнялся сценарий, а не для той оболочки,

из которой он был запущен. Команду . (точка) или source можно использовать для запуска любого сценария оболочки, а не только файла запуска, но с возможными нежелательными побочными эффектами (такими как изменения в значениях переменных оболочки, от которых зависит ваша работа). Дополнительные сведения даны в главе 10, в разделе «Локальность переменных» на с. 454.

В следующем примере в файле .bashrc устанавливается значение нескольких переменных, в том числе и PS1, переменной приглашения, чтобы в приглашении фигурировало имя хоста. Встроенная утилита . (точка) вводит новые значения в действие.

```
$ cat ~/.bashrc
export TERM=vt100               # установка типа терминала
export PS1="$(hostname -f): "   # установка строки приглашения
export CDPATH=:$HOME            # добавление HOME к строке CDPATH
stty kill '^u'                  # установка строки kill на control-u
$ . ~/.bashrc
bravo.example.com:
```

Команды, представленные символами

В оболочке Bourne Again Shell символы (,), [,] и $ используются множеством способов. Чтобы свести неразбериху к минимуму, в табл. 8.1 приводится список наиболее часто встречающихся вариантов использования каждого из этих символов. Некоторые из них в оставшейся части книги встречаться не будут.

Таблица 8.1. Встроенные команды, представленные символами

Символ	Команда
( )	Подчиненная оболочка (см. раздел «( ) средство и группировки команд, с. 298)
$( )	Замена команды (см. раздел «Командная подстановка», с. 358)
(( ))	Арифметическое вычисление; синоним встроенной команды let (используется в случае, если вложенное значение имеет знак равенства, см. раздел «Арифметические вычисления» на с. 480)
$(( ))	Арифметическое расширение (не используется, если имеется вложенный знак равенства, см. раздел «Арифметическое расширение»)
[ ]	Команда test (см. раздел «Встроенная команда test» а с. 417 и раздел «[ ] — синоним команды test» на с. 420, а также часть V)
[[ ]]	Условное выражение; подобно [ ], но добавляет сравнения строк (см. раздел «Логические вычисления (Условные выражения)» на с. 482

Перенаправление стандартной ошибки

В главе 5 было рассмотрено понятие стандартной ошибки и дано объяснение, как перенаправить стандартный вывод команды. Так, в дополнение к стандартному выводу, команды могут отправлять выход на стандартную ошибку. Команда может отправить сообщения об ошибках на стандартную ошибку, чтобы уберечь эти сообщения от перемешивания с информацией, отправляемой на стандартный вывод.

Точно так же, как это делалось со стандартным выводом, по умолчанию оболочка направляет стандартную ошибку на экран. Пока вы не перенаправите какое-то из

них, вы можете не знать, в чем разница между выводом, отправляемым командой на стандартный вывод, и выводом, отправляемым ею на стандартную ошибку. В этом разделе рассматривается синтаксис, используемый оболочкой Bourne Again Shell для перенаправления стандартной ошибки и для установки различий между стандартным выводом и стандартной ошибкой. Если вы пользуетесь оболочкой TC Shell, обратитесь к разделу «Перенаправление стандартной ошибки» на с. 377.

Описатели файлов

Описатель файла (дескриптор) — это место, куда программа отправляет свой вывод и откуда она получает свой вход. При выполнении программы Linux открывает для программы три описателя файла: 0 (стандартный ввод), 1 (стандартный вывод) и 2 (стандартная ошибка). Символ перенаправления (>, см. раздел «Перенаправление стандартного вывода» на с. 146) является сокращением от команды 1>, предписывающей оболочке перенаправить стандартный вывод. Точно так же символ < (см. раздел «Перенаправление стандартного ввода» на с. 147) является сокращением команды 0<, которая перенаправляет стандартный ввод. Символы 2> перенаправляют стандартную ошибку. Дополнительные сведения даны в разделе «Описатели файлов» на с. 499.

В следующих примерах демонстрируется, как перенаправить стандартный вывод и стандартную ошибку в разные файлы и в один и тот же файл. При запуске утилиты cat с именем несуществующего файла и с именем существующего файла эта утилита отправляет сообщение об ошибке на стандартную ошибку и копирует существующий файл в стандартный вывод. Пока они не перенаправлены, оба сообщения появляются на экране.

```
$ cat y
This is y.                          Это y
$ cat x
cat: x: No such file or directory   Файл или каталог с таким именем не существует

$ cat x y
cat: x: No such file or directory   Файл или каталог с таким именем не существует
This is y.                          Это y
```

При перенаправлении стандартного вывода команды на вывод стандартной ошибки это не влияет, и он по-прежнему появляется на экране.

```
$ cat x y > hold
cat: x: No such file or directory   Файл или каталог с таким именем не существует
$ cat hold
This is y.                          Это y
```

Точно так же, когда вы отправляете стандартный вывод через канал, это не влияет на стандартную ошибку. В следующем примере стандартный вывод команды cat отправляется через канал команде tr, которая в данном примере осуществляет преобразование символов нижнего регистра в символы верхнего регистра. (Дополнительные сведения о команде tr даны в части V.) Текст, который утилита cat отправляет на стандартную ошибку, не подвергается преобразованию, поскольку идет непосредственно на экран, а не через канал.

```
$ cat x y | tr "[a-z]" "[A-Z]"
cat: x: No such file or directory   Файл или каталог с таким именем не существует
THIS IS Y.
```

В следующем примере стандартный вывод и стандартная ошибка перенаправляются в разные файлы. Лексема, следующая за 2>, сообщает оболочке, куда перенаправить стандартную ошибку (файловый описатель 2). Лексема, следующая за 1>, сообщает оболочке, куда перенаправить стандартный вывод (файловый описатель 1). Вместо 1> можно использовать >.

```
$ cat x y 1> hold1 2> hold2
$ cat hold1
This is y.                          Это y
$ cat hold2

cat: x: No such file or directory Файл или каталог с таким именем не существует
```

Объединение стандартного вывода и стандартной ошибки

В следующем примере обозначение &> перенаправляет стандартный вывод и стандартную ошибку в один и тот же файл. Обозначение >& осуществляет то же самое действие при работе с оболочкой tcsh (см. раздел «Перенаправление стандартной ошибки» на с. 377).

```
$ cat x y &> hold
$ cat hold
cat: x: No such file or directory Файл или каталог с таким именем не существует
This is y.                          Это y
```

Дублирование описателя файла

В следующем примере первая команда 1> перенаправляет стандартный вывод в файл hold, а затем вторая команда 2>&1 объявляет описатель файла 2 дубликатом описателя файла 1. В результате и стандартный вывод, и стандартная ошибка перенаправляются в файл hold.

```
$ cat x y 1> hold 2>&1
$ cat hold

cat: x: No such file or directory Файл или каталог с таким именем не существует
This is y.                          Это y
```

В данном случае команде 1> hold предшествует команда 2>&1. Если бы они следовали в обратном порядке, стандартная ошибка была бы превращена в дубликат стандартного вывода до того, как стандартный вывод был перенаправлен в файл hold. При таком сценарии в файл hold был бы перенаправлен только стандартный вывод.

В следующем примере описатель файла 2 объявляется дубликатом описателя файла 1, а вывод для описателя файла 1 отправляется через канал команде tr.

```
$ cat x y 2>&1 | tr "[a-z]" "[A-Z]"
CAT: X: NO SUCH FILE OR DIRECTORY ФАЙЛ ИЛИ КАТАЛОГ С ТАКИМ ИМЕНЕМ НЕ СУЩЕСТВУЕТ

THIS IS Y.                          ЭТО Y
```

Перенаправление сообщений обошибках на стандартную ошибку

Для перенаправления стандартного вывода команды на стандартную ошибку можно воспользоваться командой 1>&2. Сценарии оболочки используют эту технологию для отправки вывода команды echo на стандартную ошибку. В следующем сценарии стандартный вывод первой команды echo перенаправляется на стандартную ошибку:

```
$ cat message_demo
echo Это сообщение об ошибке. 1>&2
echo Это сообщение не об ошибке.
```

Если перенаправить стандартный вывод сценария message_demo, то сообщения об ошибках (вроде того, которое выводится первой командой echo) появятся на экране, поскольку вы не перенаправили стандартную ошибку. Поскольку стандартный вывод сценария оболочки часто перенаправляется в другой файл, этой технологией можно воспользоваться для отображения на экране любого сообщения об ошибке. Эта технология используется в сценарии lnks (см. раздел «Сценарий lnks» на с. 425). Для создания дополнительных описателей файлов и перенаправления стандартного ввода, стандартного вывода и стандартной ошибки сценария оболочки из самого сценария можно также использовать встроенную утилиту exec (см. раздел «exec: Выполнение команды или перенаправление описателей файлов» на с. 469).

Оболочка Bourne Again Shell поддерживает операции перенаправления, показанные в табл. 8.2.

Таблица 8.2. Операции перенаправления

Оператор	Значение
< имя_файла	Перенаправляет стандартный ввод из файла с именем *имя_файла*.
> имя_файла	Перенаправляет стандартный вывод в файл с именем *имя_файла*, если только файл с таким именем существует и не установлено предотвращение перезаписи файлов noclobber (см. раздел «noclobber: Предотвращение перезаписи файлов» на с. 148). Если свойство noclobber не установлено, такое перенаправление создает (или перезаписывает существующий) файл с именем *имя_файла*
>| имя_файла	Перенаправляет стандартный вывод в файл с именем *имя_файла*, даже если файл существует и установлено свойство noclobber (см. раздел «noclobber: Предотвращение перезаписи файлов» на с. 148)
>> имя_файла	Перенаправляет и добавляет стандартный вывод к файлу с именем *имя_файла*, если только файл с таким именем существует и не установлено предотвращение перезаписи файлов noclobber (см. раздел «noclobber: Предотвращение перезаписи файлов» на с. 148). Если свойство noclobber не установлено, это перенаправление создает файл с именем *имя_файла*, если он не существует
&> имя_файла	Перенаправляет стандартный вывод и стандартную ошибку в файл с именем *имя_файла*
<&m	Дублирует стандартный ввод из описателя файла *m* (см. раздел «Дублирование описателя файла» на с. 450)
[n] >&m	Дублирует стандартный вывод или описатель файла *n*, если он указан, из описателя файла *m* (см. раздел «Дублирование описателя файла» на с. 450)
[n]<&–	Закрывает стандартный ввод или описатель файла *n*, если он указан (см. раздел «Дублирование описателя файла» на с. 450)
[n] >&–	Закрывает стандартный вывод или описатель файла *n*, если он указан

Написание простого сценария оболочки

Сценарий оболочки — это файл, содержащий команды, выполняемые оболочкой. В качестве команд в сценарии оболочки могут использоваться любые команды, которые можно ввести в ответ на приглашение оболочки. Например, команда в сценарии оболочки может запустить утилиту Linux, скомпилированную программу или другой сценарий оболочки. Как и команды, которые вводятся в командную строку, команды в сценарии оболочки могут использовать нестрогие ссылки на файлы и могут иметь

перенаправленный в файл или направленный через канал стандартный ввод и стандартный вывод. Можно также использовать каналы и перенаправления с вводом и выводом самого сценария.

Кроме команд, которые обычно используются в командной строке, в сценариях оболочки широко используются *команды управления процессом выполнения сценария* (также называемые *управляющими структурами*). Эта группа команд позволяет изменять порядок выполнения команд в сценарии, точно так же, как это делается с порядком выполнения инструкций в структурированных языках программирования. Особенности этих команд приводятся в в разделе «Управляющие структуры» (для bash) на с. 417, и в разделе с таким же названием в главе 9 (для tcsh). Оболочка интерпретирует и выполняет одну за другой команды, имеющиеся в сценарии оболочки. Таким образом, сценарий оболочки позволяет просто и быстро запускать сложные серии задач или повторяющиеся процедуры.

chmod: Превращение файла в исполняемый

Чтобы выполнить сценарий оболочки, задав его имя в качестве команды, у вас должно быть разрешение на чтение и выполнение файла, в котором содержится этот сценарий (см. главу 4, раздел «Права доступа»). Право на чтение позволяет читать файл, в котором содержится сценарий. Право на выполнение сообщает оболочке и системе, что владелец, группа и (или) все остальные пользователи имеют права на выполнение файла (подразумевается, что содержимое файла может быть выполнено).

Когда сценарий оболочки создается с помощью редактора, файл обычно не имеет установок на разрешение его выполнения. В следующем примере показан файл по имени whoson, в котором содержится сценарий оболочки:

```
$ cat whoson
date
echo "Users Currently Logged In" Пользователи, вошедшие в систему
who
$ ./whoson
bash: ./whoson: Permission denied     Отсутствие прав доступа
```

Вы не можете выполнить файл whoson, задав его имя в виде команды, поскольку для этого файла у вас нет права на выполнение. Оболочка не распознает файл whoson как исполняемый, а при попытке его выполнить выдает сообщение об ошибке Permission denied (Отсутствие прав доступа). (См. далее совет о том, что делать, если вы получаете сообщение об ошибке «command not found», то есть «команда не найдена».) Когда в качестве аргумента оболочке bash дается имя файла (bash whoson), bash воспринимает аргумент как сценарий оболочки и выполняет его. В данном случае bash является выполняемой программой, а whoson является аргументом, который bash выполняет, поэтому для файла whoson не нужны права на его выполнение. Но у вас должно быть право на чтение. То же самое можно сделать и с файлами сценариев оболочки tcsh.

Утилита chmod изменяет права доступа, связанные с файлом. На рис. 8.1 показана команда ls вызываемая с ключом -l, которая отображает права доступа, установленные для файла whoson до и после того, как утилита chmod предоставила владельцу файла доступ на выполнение. Первая команда ls отображает на месте четвертого символа

дефис (-), свидетельствующий о том, что у владельца нет права на выполнение файла. Затем утилита chmod предоставляет владельцу право на выполнение: аргумент u+x заставляет chmod добавить (+) право на выполнение (x) для владельца (u). (Буква u означает *user* — пользователь, хотя это в данном случае относится не к любому пользователю, а только к владельцу файла.) В качестве второго аргумента используется имя файла. Вторая команда ls показывает в четвертой позиции букву x, свидетельствующую о том, что владелец обладает правами выполнения.

```
$ ls -l whoson
-rw-rw-r--   1 max group 40 May 24 11:30 whoson

$ chmod u+x whoson
$ ls -l whoson
-rwxrw-r--   1 max group 40 May 24 11:30 whoson

$ ./whoson
Fri May 22 11:40:49 PDT 2009
Users Currently Logged In
zach     pts/7    May 21 18:17
hls      pts/1    May 22 09:59
sam      pts/12   May 22 06:29 (bravo.example.com)
max      pts/4    May 22 09:08
```

Рис. 8.1. Использование chmod для превращения сценария оболочки в исполняемый файл

КОМАНДА НЕ НАЙДЕНА — COMMAND NOT FOUND?

Если имя файла сценария оболочки дано как команда и в него не включен префикс ./, оболочка обычно выводит следующее сообщение об ошибке:

```
$ whoson
bash: whoson: command not found   (команда не найдена)
```

Это сообщение свидетельствует о том, что оболочка не настроена на поиск исполняемых файлов в рабочем каталоге. Теперь введите следующую команду:

```
$ ./whoson
```

Префикс ./ в явном виде сообщает оболочке, что нужно искать исполняемый файл в рабочем каталоге. Чтобы изменить среду окружения, заставляя оболочку искать файл в рабочем каталоге, изучите раздел, посвященный переменной PATH.

Если файл будет выполняться другими пользователями, нужно также изменить групповые права доступа к файлу и (или) права для всех остальных пользователей. Чтобы имя файла использовалось как команда, права выполнения должны быть у любого пользователя. Если файл является сценарием оболочки, то пользователь, пытающийся выполнить файл, должен также иметь право на чтение этого файла. Для выполнения двоичных файлов (скомпилированных программ) наличие права на чтение не требуется.

Заключительная команда, показанная на рис. 8.1, демонстрирует оболочку, выполняющую файл, имя которого было задано в качестве команды. Дополнительные сведения даны в главе 4, в разделе «Права доступа», а также в части V (в материалах, посвященных утилитам ls и chmod).

#! — способ указания оболочки

В первой строке сценария оболочки можно указать специальную последовательность символов, которая сообщит операционной системе, какая оболочка (или другая программа) должна выполнить файл. Поскольку операционная система проверяет начальные символы программы до того, как попытаться выполнить его с помощью команды exec, эти символы уберегают систему от неудачной попытки. Если первыми двумя символами сценария будут #!, система истолкует следующие за ними символы как абсолютное имя утилиты, которая должна выполнить сценарий. Это может быть имя не только оболочки, но и любой другой программы. В следующем примере указывается, что сценарий должна запустить оболочка bash:

```
$ cat bash_script
#!/bin/bash
echo "This is a Bourne Again Shell script."
```

Это сценарий для оболочки Bourne Again Shell

Символы #! можно применить в том случае, когда у вас есть сценарий, который нужно запустить в оболочке, отличающейся от той, в которой он запускается. В следующем примере показан сценарий, который должен быть выполнен оболочкой tcsh:

```
$ cat tcsh_script
#!/bin/tcsh
echo "This is a tcsh script."          Это сценарий для оболочки tcsh
set person = zach
echo "person is $person"
```

Благодаря наличию строки #! операционная система гарантирует, что tcsh выполнит сценарий независимо от того, из какой оболочки он запускался.

Внутри сценария можно использовать команду ps -f для вывода имени оболочки, выполняющей сценарий. В трех строках, которые выводятся командой ps в следующем примере, показан процесс, выполняющий родительскую оболочку bash, процесс, выполняющий сценарий оболочки tcsh, и процесс, выполняющий команду ps:

```
$ cat tcsh_script2
#!/bin/tcsh
ps -f

$ ./tcsh_script2
UID        PID  PPID  C STIME TTY          TIME CMD
max       3031  3030  0 Nov16 pts/4    00:00:00 -bash
max       9358  3031  0 21:13 pts/4    00:00:00 /bin/tcsh ./tcsh_script2
max       9375  9358  0 21:13 pts/4    00:00:00 ps -f
```

Если за символами #! не поставить имя исполняемой программы, оболочка сообщит, что он не может найти команду, которая запрошена на запуск. Можно дополнительно поставить за символами #! несколько пробелов. Если строка с символами #! будет опущена и будет предпринята попытка запустить, к примеру, сценарий для tcsh из оболочки bash, сценарий запустится под оболочкой bash, что может привести к генерации сообщений об ошибках, или же сценарий будет неправильно работать. Пример автономного сценария для оболочки sed, в котором используется строка #!, показан в разделе «Автономный сценарий» на с. 597.

— символ начала комментария

Комментарии упрощают чтение и сопровождение вами и другими специалистами сценариев оболочки и всего кода. Для обеих оболочек, Bourne Again Shell и TC Shell, используется общий синтаксис комментариев. Если знак решетки (#) стоит в первой позиции первой строки сценария, а сразу же за ним не следует восклицательный знак (!), или если знак решетки встречается где-нибудь в другом месте сценария, оболочка воспринимает его как начало комментария. Оболочка игнорирует все, что находится между знаком решетки и концом строки (следующим символом новой строки).

Выполнение сценария оболочки

Системные вызовы fork и exec

Команды, запускаемые в командной строке, заставляют оболочку создавать ответвление на новый процесс, создавая дубликат процесса оболочки (подчиненную оболочку). Новый процесс пытается выполнить команду. Как и разветвление fork, подпрограмма выполнения exec выполняется операционной системой (путем системного вызова). Если команда является двоичной исполняемой программой, например скомпилированной программой на языке C, системный вызов exec будет выполнен, и система загрузит поверх только что созданной оболочки исполняемую программу. Если команда является сценарием оболочки, exec потерпит неудачу. Когда произойдет отказ exec, будет сделано предположение, что команда является сценарием оболочки, и подчиненная оболочка запустит имеющиеся в сценарии команды. В отличие от оболочки, предназначенной для входа в систему, которая ожидает ввода из командной строки, подчиненная оболочка получает ввод из файла, а именно из сценария оболочки.

Как уже ранее отмечалось, вы можете запускать команды, имеющиеся в файле сценария оболочки, на который у вас нет прав выполнения, путем использования команды bash для системного вызова exec и запуска оболочки, которая непосредственно запустит сценарий. В следующем примере оболочка bash создает новую оболочку, которая получает вводную информацию из файла по имени whoson:

```
$ bash whoson
```

Поскольку команда bash ожидает чтения файла, содержащего команды, вам не нужно иметь право на выполнение файла whoson. (Но нужно иметь право на его чтение.) Но даже при том, что bash читает и выполняет команды, имеющиеся в whoson, стандартный ввод, стандартный вывод и стандартная ошибка остаются направленными из терминала и в него.

Несмотря на то что bash может использоваться для выполнения сценария оболочки, эта технология приводит к тому, что сценарий запускается медленнее, чем тогда, когда вы сами присваиваете себе право на выполнение его файла и вызываете сценарий непосредственным образом. Обычно пользователи предпочитают превратить файл в исполняемый и запустить сценарий, набрав его имя в командной строке. К тому же имя набрать проще, тем более что это действие соответствует способу вызова других видов программ (то есть не нужно знать, что именно запускается, сценарий оболочки или исполняемый файл). Тем не менее, если bash не является используемой вами интерактивной оболочкой, или если вы хотите посмотреть, как сценарий работает с другими оболочками, может понадобиться запустить сценарий, указав имя его файла в качестве аргумента для bash или tcsh.

Разделение и группировка команд

Независимо от того, вводите ли вы команды в оболочку в интерактивном режиме, или пишите сценарий оболочки, вы должны отделять команды друг от друга. В этом разделе, который касается оболочек Bourne Again Shell и TC Shell, рассматриваются способы разделения команд, которые уже приводились в главе 5, и представляются *несколько новых способов.*

Точка с запятой (;) и перевод строки (↵) разделяют команды

Разделитель строк является уникальным средством разделения команд, поскольку этот символ инициирует выполнение предшествующей ему команды. Вы наблюдали подобное поведение при разборе всего предыдущего материала этой книги, когда нажимали клавишу ВВОД, заканчивая набор команды в командной строке.

Точка с запятой (;) является разделителем команд, который не инициирует выполнение команды и не изменяет никаких аспектов ее функционирования. Можно последовательно выполнить целый ряд команд путем ввода их в одну и ту же командную строку и отделения каждой команды от последующей с помощью точки с запятой (;). Выполнение последовательности команд инициируется нажатием клавиши ВВОД:

```
$ x ; y ; z
```

Если x, y и z являются командами, то предыдущая командная строка приведет к тем же результатам, что и следующие три команды. Разница состоит в том, что в следующем примере оболочка выдает приглашение после каждой команды (x, y и z), завершая выполнение, а предыдущая командная строка заставляет оболочку выдать приглашение только после завершения выполнения команды z:

```
$ x
$ y
$ z
```

Пробельный символ

Хотя пробельные символы вокруг точек с запятыми упрощали чтение команд, их присутствия не требуется. Никакие разделители команд не нуждаются в окружении пробелов или знаков табуляции.

\ — знак продолжения команды

Когда вводится длинная командная строка и курсор приближается к правому краю экрана, можно продолжить команду на следующей строке, воспользовавшись символом обратного слэша (\). Символ обратного слэша цитирует или выключает разделитель строк, который за ним следует, поэтому оболочка не рассматривает разделитель строк как символ завершения команды. Если обратный слэш заключить в одинарные кавычки или поставить перед ним еще один обратный слэш, то он утратит свои цитирующие свойства в отношении специальных символов, в том числе и в отношении разделителя строк (но не в tcsh; см. раздел «prompt2» на с. 392). Заключение обратного слэша в двойные кавычки не лишает его цитирующих свойств (но не в tcsh).

Хотя строку можно разбить и в середине слова (лексемы), но обычно проще сделать это сразу же перед или после пробельного символа.

В bash в командной строке можно вставить ВВОД в середине заключенной в кавычки строки и обойтись без использования обратного слэша. (Как при этом ведет себя оболочка tcsh, рассмотрено в разделе «prompt2» на с. 392). При этом введенный вами символ разделения строк (генерируемый клавишей ВВОД) станет частью строки:

```
$ echo "Please enter the three values
> required to complete the transaction."
Please enter the three values required to complete the transaction.
```

В трех примерах, приводимых в данном разделе, bash не интерпретирует ВВОД как окончание команды, поскольку генерируемый этой клавишей символ появляется внутри строки, заключенной в кавычки. Знак «больше» (>) является вторичным приглашением (см. раздел «Переменная PS2: Приглашение пользователю (вторичное)» на с. 315), показывающим, что оболочка ждет от пользователя продолжения ввода незавершенной команды. В следующем примере первый ВВОД процитирован (отключен), поэтому оболочка рассматривает обратный слэш как разделитель и не интерпретирует его буквально.

```
$ echo "Please enter the three values \
> required to complete the transaction."
Please enter the three values required to complete the transaction.
```

Одинарная кавычка заставляет оболочку интерпретировать обратный слэш буквально:

```
$ echo 'Please enter the three values \
> required to complete the transaction.'
Please enter the three values \
required to complete the transaction.
```

| и & — разделяют команды, но не только

Символ канала (|) и символ фоновой задачи (&) также являются разделителями команд. Они не инициируют выполнение команды, но изменяют некоторые аспекты функционирования команды. Символ канала меняет источник стандартного ввода или получатель стандартного вывода. Символ фоновой задачи заставляет оболочку выполнять задачу в фоновом режиме и незамедлительно вывести приглашение, позволяя продолжить работу над другими задачами.

Каждая из следующих командных строк инициирует отдельное задание, включающее три задачи:

```
$ x | y | z
$ ls -l | grep tmp | less
```

В первом задании оболочка перенаправляет стандартный вывод задачи x на стандартный ввод задачи y и перенаправляет стандартный вывод задачи y, на стандартный ввод задачи z. Поскольку все задание запускается на выполнение в активном режиме, оболочка не выводит приглашение до тех пор, пока не будет выполнена до конца задача z:, а задача z не будет завершена, пока не завершится задача y, которая, в свою очередь, не будет завершена, пока не завершится задача x. Во втором задании задачей x является команда `ls -l`, задачей y — команда `grep tmp`, а задачей z — разбиватель на страницы, команда `less`. Оболочка выводит длинный (и широкий) листинг файлов в рабочем каталоге, в которых содержится строка `tmp`, пропуская его через канал к команде `less`. В следующей командной строке задачи d и e выполняются в фоновом, а задача f выполняется в активном режиме:

```
$ d & e & f
[1] 14271
[2] 14272
```

В квадратных скобках оболочка выводит номер задания, за которым следует PID-номер каждого процесса, запущенного в фоновом режиме. Как только выполнение задачи f будет завершено, оболочка выведет приглашение. Это может произойти еще до того, как будет завершено выполнение задачи d или задачи e.

Перед выводом приглашения на ввод новой команды оболочка проверяет, не завершилось ли какое-нибудь из фоновых заданий. Для каждого завершившегося задания оболочка выводит его номер и слово Done (Выполнено), а также командную строку, вызвавшую выполнение задания, после чего оболочка выводит приглашение. При выводе списка номеров заданий за номером последнего запущенного задания следует символ +, а за номером предыдущего задания следует символ -.

За номерами других заданий следуют пробелы. После запуска последней команды оболочка перед выдачей приглашения выводит следующие строки:

```
[1]-  Done                    d
[2]+  Done                    e
```

В следующей командной строке все три задачи выполняются в фоновом режиме, и оболочка выдает приглашение незамедлительно:

```
$ d & e & f &
[1] 14290
[2] 14291
[3] 14292
```

Канал можно использовать для отправки выходной информации из одной задачи следующей задаче, а знак амперсанда (&) можно использовать для запуска всего задания в виде фоновой задачи. В этом случае оболочка опять выдаст приглашение незамедлительно. Оболочка относится к командам, объединенным каналом, как к единому заданию. То есть она рассматривает все каналы как единые задания, не важно, сколько задач связано символом канала (|) или насколько сложными они являются. Оболочка Bourne Again Shell выдает отчет только об одном процессе, запущенном в фоновом режиме (хотя на самом деле таких процессов три):

```
$ d | e | f &
[1] 14295
```

Оболочка TC Shell показывает запущенными в фоновом режиме три процесса (относя их принадлежность к заданию 1):

```
tcsh $ d | e | f &
[1] 14302 14304 14306
```

( ) — средство группировки команд

Для группировки команд можно воспользоваться круглыми скобками. Для каждой группы оболочка создает свою собственную копию, называемую *подчиненной оболочкой*. Каждая группа команд рассматривается оболочкой в качестве задания, и для выполнения каждой команды создается новый процесс. (Дополнительные сведения о создании подчиненных оболочек даны в разделе «Структура процессов» на с. 321.) Каждая подчиненная оболочка (задание) имеет свою собственную среду окружения, это значит, что у нее есть свой собственный набор переменных, чьи значения могут отличаться от значений таких же переменных других подчиненных оболочек.

Следующая командная строка выполняет команды a и b последовательно в фоновом режиме, одновременно с выполнением в фоновом режиме команды c. При этом оболочка тут же выдает приглашение.

```
$ (a ; b) & c &
[1] 15520
[2] 15521
```

Предыдущий пример отличается от более раннего примера `d & e & f &` тем, что задачи `a` и `b` инициируются последовательно, а не параллельно.

Подобно этому, следующая командная строка выполняет `a` и `b` последовательно в фоновом режиме и в то же самое время выполняет последовательно `c` и `d`, и тоже в фоновом режиме. Одна подчиненная оболочка, выполняющая `a` и `b`, и другая подчиненная оболочка, выполняющая `c` и `d`, работают параллельно. Приглашение оболочка выдает без промедления.

```
$ (a ; b) & (c ; d) &
[1] 15528
[2] 15529
```

Следующий сценарий копирует один каталог в другой. Вторая пара скобок создает подчиненную оболочку для запуска команд, следующих за символом канала. Благодаря этим скобкам вывод первой команды tar становится доступным второй команде tar, несмотря на промежуточную команду cd. Если бы скобок не было, вывод первой команды tar отправлялся бы команде cd и был бы потерян, поскольку cd не обрабатывает ввод из стандартного ввода.

Переменные оболочки $1 и $2 представляют собой соответственно первый и второй аргументы командной строки (см. раздел «$1–$n: Аргументы командной строки» на с. 460). Первая пара скобок, создающая подчиненную оболочку для запуска первых двух команд, позволяет пользователям вызывать сценарий cpdir с относительными именами. Без этих скобок первая команда cd изменила бы рабочий каталог сценария (а следовательно, и рабочий каталог второй команды cd). Благодаря скобкам изменяется только рабочий каталог подчиненной оболочки.

```
$ cat cpdir
(cd $1 ; tar -cf - . ) | (cd $2 ; tar -xvf - )
$ ./cpdir /home/max/sources /home/max/memo/biblio
```

Командная строка cpdir копирует файлы и каталоги, имеющиеся в каталоге /home/max/sources, в каталог по имени /home/max/memo/biblio. Этот сценарий оболочки работает почти так же, как утилита cp с ключом -r. Дополнительные сведения об утилитах cp и tar даны в части V.

Управление заданиями

Задание представляет собой конвейер команд. При каждом вводе команды в оболочку вы запускаете простое задание. Например, если вы набрали в командной строке дату и нажали ВВОД, вы запустили задание. Можно также создать в командной строке несколько заданий, состоящих из нескольких команд:

```
$ find . -print | sort | lpr & grep -l max /tmp/* > maxfiles &
[1] 18839
[2] 18876
```

Та часть командной строки, которая расположена до первого символа &, станет одним заданием, состоящим из трех процессов, связанных каналами: find (см. часть V), sort (см. раздел «sort: Отображение содержимого файла в определенном порядке» на с. 74 и часть V) и lpr (см. раздел «lpr: Распечатка файла» на с. 71 и часть V). Второе задание состоит из одного процесса, выполняющего команду grep. Замыкающие символы & помещают каждое задание в фоновый режим, поэтому оболочка bash не дожидается их завершения, перед тем как вывести приглашение.

Используя управление заданиями, вы можете перемещать команды из активного режима в фоновый (и обратно), временно останавливать выполнение команд и выводить список всех запущенных в фоновом режиме или остановленных команд.

jobs: Вывод списка заданий

Встроенная команда jobs выводит список всех заданий, выполняющихся в фоновом режиме. В следующем примере команда sleep запускается в фоновом режиме и создает фоновое задание, о котором рапортует команда jobs:

```
$ sleep 60 &
[1] 7809
$ jobs
[1] + Running                 sleep 60 &
```

fg: Перевод задания в активный режим

Каждой команде, запущенной в фоновом режиме, оболочка присваивает номер задания. Для каждого задания, выполняющегося в фоновом режиме, оболочка тут же, перед тем как вывести приглашение, выводит номер задания и PID-номер:

```
$ xclock &
[1] 1246
$ date &
[2] 1247
$ Tue Dec 2 11:44:40 PST 2008
[2]+ Done          date
$ find /usr -name ace -print > findout &
[2] 1269
$ jobs
[1]- Running         xclock &
[2]+ Running         find /usr -name ace -print > findout &
```

Номера заданий, утрачивающие свое значение по окончании их выполнения, могут быть использованы повторно. Когда задание запускается в фоновом режиме или переводится в него, оболочка присваивает ему номер, который превышает на единицу самый большой из используемых номеров заданий.

В предыдущем примере команда jobs выводит в списке первое задание, xclock, как задание 1. Команда date не появляется в списке заданий, потому что оно завершается еще до того, как запускается команда jobs. Поскольку команда date была завершена до запуска команды find, команда find становится заданием 2.

Для перевода фонового задания в активный режим используется встроенная команда fg, за которой следует номер задания. Можно также в качестве команды ввести знак процента (%), за которым указать номер задания. Каждая из следующих команд переводит задание 2 в активный режим. Когда вы переводите задание в активный режим, оболочка выводит текущую команду, выполняемую в активном режиме.

```
$ fg 2
find /usr -name ace -print > findout
```

или

```
$ %2
find /usr -name ace -print > findout
```

Можно также сослаться на задание, если ввести после знака процента строку, *однозначно идентифицирующую начало командной строки, которая использовалась* для запуска задания. Вместо предыдущей команды можно было использовать либо

команду fg %find, либо команду fg %f, поскольку обе из них уникальным образом идентифицируют задание 2. Если за знаком процента поставить вопросительный знак и строку, то строка сможет соответствовать любой части командной строки. В предыдущем примере команда fg %?ace также переведет задание 2 в активный режим.

Зачастую задание, которое нужно перевести в активный режим, является единственным, запущенным в фоновом режиме, или заданием, которое команда jobs показывает в списке со знаком плюс (+). В таких случаях команда fg без аргументов переводит задание в активный режим.

Приостановка задания

При нажатии клавиши приостановки (обычно это CONTROL+Z) задание, выполняемое в активном режиме, немедленно приостанавливается (временно останавливается), и выводится сообщение, в которое включается слово Stopped (Остановлено).

```
CONTROL+Z
[2]+  Stopped                 find /usr -name ace -print > findout
```

Дополнительные сведения даны в разделе «Перевод задания из активного в фоновый режим» на с. 155.

bg: Отправка задания в фоновый режим

Чтобы перевести в фоновый режим задание, выполняемое в активном режиме, нужно сначала приостановить выполнение задания (см. выше). Затем можно воспользоваться встроенной командой bg для возобновления выполнения задания в фоновом режиме.

```
$ bg
[2]+ find /usr -name ace -print > findout &
```

Если фоновое задание пытается что-нибудь прочитать с терминала, оболочка останавливает программу и выводит сообщение, в котором говорится, что задание приостановлено. После этого вы должны перевести задание в активный режим, чтобы оно смогло прочитать данные с терминала.

```
$ (sleep 5; cat > mytext) &
[1] 1343
$ date
Tue Dec  2 11:58:20 PST 2008
[1]+ Stopped                  ( sleep 5; cat >mytext )
$ fg
( sleep 5; cat >mytext )
Remember to let the cat out!          Не забудьте выпустить наружу утилиту cat!
CONTROL+D
$
```

В предыдущем примере оболочка выводит номер задания и PID-номер фонового задания, как только оно будет запущено, после чего выводится приглашение. Чтобы показать, что в данный момент можно вводить команду, пользователь вводит команду date, и ее вывод появляется на экране. Оболочка ждет и непосредственно перед выводом сообщения (после окончания выполнения команды date) уведомляет вас о приостановке задания 1. Когда вы вводите команду fg, оболочка переводит задание

в активный режим, и вы можете ввести данные, которые ожидаются командой. В данном случае ввод нужно закончить комбинацией CONTROL+D, которая отправляет оболочке сигнал EOF (end of file — конец файла. После этого оболочка выводит новое приглашение.

Оболочка информирует вас об изменениях статуса задания, уведомляя, когда задание запускается в фоновом режиме, когда оно завершается или останавливается (возможно, из-за ожидания ввода с терминала). Оболочка также оповещает вас, когда задание, выполняемое в фоновом режиме, приостанавливается. Поскольку оповещения о заданиях, выполняемых в фоновом режиме, могут помешать вашей работе, оболочка откладывает вывод этих оповещений до того момента, когда она может вывести приглашение. Можно настроить свойство notify (табл. 8.13), которое заставит оболочку выводить такие оповещения без задержки.

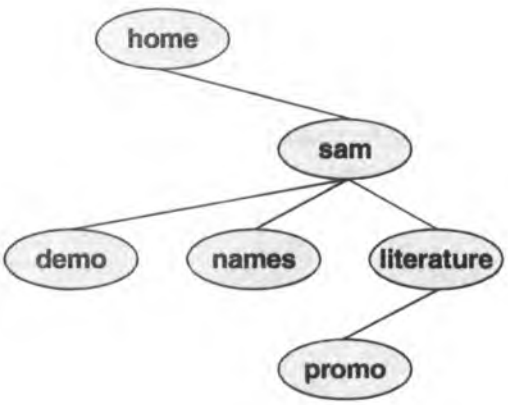

Рис. 8.2. Структура каталогов, используемых в примерах

Если вы пытаетесь выйти из оболочки при приостановленном задании, оболочка выведет предупреждение и не позволит вам выйти из нее. Если после этого вы воспользуетесь командой jobs для просмотра списка заданий или тут же еще раз попытаетесь выйти из оболочки, то она позволит вам это сделать. Если в соответствии с установками по умолчанию свойство huponexit (см. табл. 8.13) не будет установлено, выход не состоится, и фоновые задания будут и дальше выполняться в фоновом режиме. Если это свойство будет установлено, оболочка прекратит выполнение всех заданий.

Работа со стеком каталогов

Обе оболочки, и Bourne Again Shell, и TC Shell, позволяют сохранять список каталогов, с которыми вы работаете, давая возможность простого перемещения между ними. На этот список ссылаются как на *стек*. Структура стека аналогична стопке кухонных тарелок: вы ставите очередную тарелку на вершину стопки и забираете ее оттуда же. Такой тип стека называется «последним пришел — первым ушел» — LIFO (last in, first out).

dirs: Отображение стека

Встроенная команда dirs отображает содержимое стека каталогов. Когда команда dirs вызывается при пустом стеке, она отображает имя рабочего каталога:

```
$ dirs
~/literature
```

Для представления имени личного каталога пользователя во встроенной команде dirs используется тильда (~). Примеры в нескольких следующих разделах предполагают, что ссылки делаются на структуру каталогов, показанную на рис. 8.2.

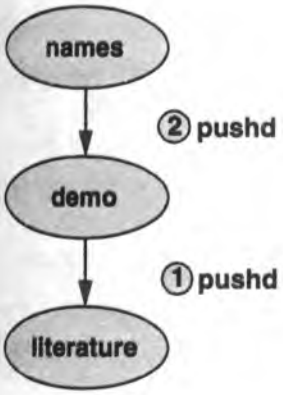

Рис. 8.3. Создание стека каталогов

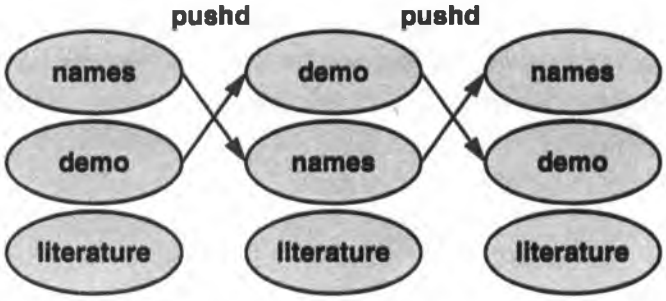

Рис. 8.4. Использование команды pushd для смены рабочих каталогов

pushd: Помещение каталога в стек

Когда встроенной команде pushd (push directory — поместить каталог) предоставляется один аргумент, она помещает каталог, указанный в аргументе, в стек, меняет каталоги на указанный каталог и отображает стек. Работа следующего примера проиллюстрирована на рис. 8.3.

```
$ pushd ../demo
~/demo ~/literature
$ pwd
/home/sam/demo
$ pushd ../names
~/names ~/demo ~/literature
$ pwd
/home/sam/names
```

При использовании команды pushd без аргумента она меняет местами верхние два каталога в стеке, превращая новый каталог, находящийся на вершине стека (который до этого был вторым в стеке), в новый рабочий каталог и отображая стек (рис. 8.4):

```
$ pushd
~/demo ~/names ~/literature
$ pwd
/home/sam/demo
```

Такое использование команды pushd позволяет легко осуществлять переходы вперед и назад между двумя каталогами. Независимо от того, был ли в явном виде создан стек каталогов или не был, для смена рабочего каталога на предыдущий можно воспользоваться и командой `cd -`. Для доступа к другому, имеющемуся в стеке каталогу нужно вызвать команду pushd с числовым аргументом, поставив перед ним знак плюс. Нумерация каталогов в стеке начинается с верхнего каталога, который имеет номер 0. Следующая команда pushd является продолжением предыдущего примера, меняя рабочий каталог на literature и перемещая literature на вершину стека:

```
$ pushd +2
~/literature ~/demo ~/names
$ pwd
/home/sam/literature
```

popd: Удаление каталога из стека

Для удаления каталога из стека используется встроенная команда popd (pop directory — вытолкнуть каталог из стека). Как видно из следующего примера и рис. 8.5, команда popd без аргументов удаляет из стека запись о самом верхнем каталоге и превращает в рабочий тот каталог, чье имя стало самым верхнем в стеке:

```
$ dirs
~/literature ~/demo ~/names
$ popd
~/demo ~/names
$ pwd
/home/sam/demo
```

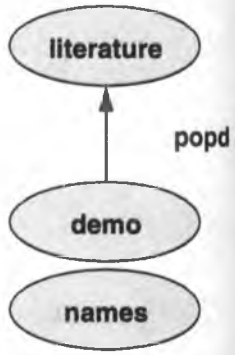

Рис. 8.5. Использование команды popd для удаления каталога из стека

Для удаления из стека не самого верхнего имени каталога команда popd используется с числовым аргументом, которому предшествует знак «+». В следующем примере удаляется имя каталога с номером 1, demo. Удаление имени каталога, чей номер отличается от нулевого, не приводит к смене рабочего каталога.

```
$ dirs
~/literature ~/demo ~/names
$ popd +1
~/literature ~/names
```

Параметры и переменные

Переменные

При работе под управлением оболочки *параметр оболочки* связан со значением, доступным пользователю. Существует несколько видов параметров оболочки. На параметры, чьи имена состоят из букв, цифр и знаков подчеркивания, часто ссылаются как на *переменные оболочки* или просто как на *переменные*. Имя переменной должно начинаться с буквы или знака подчеркивания, а не с цифры. Поэтому A76, MY_CAT и ___X___ являются допустимыми именами переменных, а 69TH_STREET (начинающееся с цифры) и MY-NAME (содержащее дефис) таковыми не являются.

Пользовательские переменные

Те переменные оболочки, которым вы даете имена и присваиваете значения, являются *пользовательскими переменными*. Значения пользовательских переменных в любой момент можно изменить или сделать их переменными *только для чтения*, чтобы их значения нельзя было изменить. Пользовательские переменные можно также сделать *глобальными*. Глобальная переменная (которая также называется *переменной окружения*) доступна всем оболочкам и другим программам, запускаемым в процессах, ответвляющихся от исходной оболочки. Существует соглашение об именах, согласно которому для имен глобальных переменных используются только буквы верхнего регистра, а для имен всех остальных переменных используется смешанный регистр или буквы нижнего регистра. Дополнительные сведения о глобальных переменных даны в разделе «Локальность переменных» на с. 454. В оболочке Bourne Again Shell для присвоения значения переменной используется следующий синтаксис:

```
ПЕРЕМЕННАЯ=значение
```

Пробелов с обоих сторон знака равенства (=) быть не должно. Вот как выглядит пример инструкции присваивания:

```
$ myvar=abc
```

При работе с оболочкой TC Shell инструкции присваивания должно предшествовать слово set, а пробелы по обе стороны от знака равенства ставить необязательно:

```
$ set myvar = abc
```

Оболочка Bourne Again Shell разрешает помещать инструкцию присваивания значения переменной в командную строку. Такой вид присваивания создает переменную с локальной областью видимости для командной оболочки, то есть эта переменная доступна только из программы, запущенной с помощью команды.

Сценарий оболочки my_script выводит значение переменной TEMPDIR. Следующая команда запускает сценарий my_script со значением переменной TEMPDIR, установленным в /home/sam/temp. Встроенная команда echo показывает, что интерактивная оболочка не содержит значения для TEMPDIR после запуска сценария my_script. Если значение TEMPDIR было установлено в интерактивной оболочке, то такой запуск сценария my_script не повлияет на значение этой переменной.

```
$ cat my_script
echo $TEMPDIR
$ TEMPDIR=/home/sam/temp ./my_script
/home/sam/temp
$ echo $TEMPDIR

$
```

Переменные, являющиеся ключевыми словами

Переменные оболочки, являющиеся ключевыми словами (или просто переменные, являющиеся ключевыми словами) имеют специальное значение для оболочки и часто имеют короткие, хорошо запоминающиеся имена. При запуске оболочки (к примеру, при входе в систему) она получает по наследству от среды окружения несколько переменных, являющихся ключевыми словами. В число этих переменных входят переменная HOME, которая идентифицирует ваш личный каталог, а также переменная PATH, которая определяет, в каких каталогах оболочка будет вести поиск и в какой последовательности. При запуске оболочка создает и присваивает значение по умолчанию другим переменным, являющимся ключевыми словами. А все остальные переменные не существуют до тех пор, пока вы не установите для них каких-либо значений.

Значение большинства переменных оболочки, являющихся ключевыми словами, можно изменить. Обычно не возникает необходимости изменять значения таких переменных, инициализированных во влияющих на всю систему файлах запуска /etc/profile или /etc/csh.cshrc. Если все-таки нужно изменить значение переменной оболочки bash или tcsh, являющейся ключевым словом, это нужно сделать в одном из ваших файлов запуска (для bash см. раздел «Файлы запуска» на с. 284 в данной главе, а для tcsh см. раздел с таким же именем на с. 393). Переменные, являющиеся ключевыми словами, можно сделать глобальными точно так же, как это делается с пользовательскими переменными — эта задача обычно выполняется автоматически в файлах запуска. Эти переменные можно также превратить в переменные только для чтения.

Позиционные и специальные параметры

Имена позиционных и специальных параметров не похожи на имена простых переменных. У большинства этих параметров односимвольные имена (например, 1,? и #) и ссылаются на них (как и на все другие переменны) предваряя имя знаком доллара ($1, $? и $#). Значения этих параметров отражают различные аспекты вашего происходящего взаимодействия с оболочкой.

При каждом вводе команды каждый аргумент командной строки становится значением *позиционного параметра* (см. раздел «Позиционные параметры» на с. 459). Позиционные параметры дают доступ к аргументам командной строки, и эта возможность будет часто востребована при написании сценариев оболочки. Встроенная команда set (см. раздел «set: Инициализация аргументов командной строки» на с. 461) позволяет присваивать значения позиционным параметрам.

Другие часто востребуемые сценариями оболочки значения, например имя последней выполненной команды, количество аргументов командной строки и состояние самой последней выполняемой команды, доступны в виде *специальных параметров* (см. раздел «Специальные параметры» на с. 457). Присвоить значения специальным параметрам невозможно.

Пользовательские переменные

В первой строке следующего примера объявляется переменная по имени person, которая инициализируется значением max (в tcsh следует использовать команду set person = max):

```
$ person=max
$ echo person
person
$ echo $person
max
```

Замена параметра

Поскольку встроенная команда echo копирует свои аргументы на стандартный вывод, ее можно использовать для вывода значений переменных. Во второй строке предыдущего примера показано, что person не означает max. Строковое значение person и выводится как person. Оболочка заменяет имя переменной ее значением, только если перед именем поставлен знак доллара ($). Команда echo $person выводит значение переменной person, а не отображает строку $person, поскольку оболочка не передает $person команде echo в качестве аргумента. Благодаря лидирующему знаку $ оболочка распознает $person в качестве имени переменной, заменяет его значением переменной и передает это значение команде echo. Эта встроенная команда выводит значение переменной, а не ее имя, даже «не зная» о том, что она была вызвана с переменной.

Цитирование знака $

Можно предотвратить проводимую оболочкой замену имени переменной ее значением, если процитировать ведущий знак $. Но двойные кавычки не предохранят от замены, здесь следует воспользоваться одинарными кавычками или обратным слэшем (\).

```
$ echo $person
max
```

```
$ echo "$person"
max
$ echo '$person'
$person
$ echo \$person
$person
```

Пробелы

Благодаря тому, что двойные кавычки не препятствуют замене имени переменной ее значением, но при этом выключают специальное значения большинства символов, их удобно применять при присваивании значений переменным и при использовании этих значений. При присваивании переменной значения, в котором содержатся пробелы или знаки табуляции, это значение нужно взять в двойные кавычки. Хотя двойные кавычки требуются не во всех случаях, их использование считается хорошей привычкой.

```
$ person="max and zach"
$ echo $person
max and zach
$ person=max and zach
bash: and: command not found      and: команда на найдена
```

При ссылке на переменную, значение которой содержит знаки табуляции или несколько непрерывных пробелов, нужно использовать кавычки, чтобы сберечь промежутки. Если этого не сделать, оболочка, перед тем как передать утилите значение переменной, сожмет каждую строку пустых символов в один пробел:

```
$ person="max    and    zach"
$ echo $person
max and zach
$ echo "$person"
max   and   zach
```

Расширение имен в присваиваниях

При выполнении команды с переменной в качестве аргумента оболочка заменяет имя переменной его значением и передает это значение выполняемой программе. Если имя переменной содержит специальный символ, например * или ?, оболочка может расширить эту переменную.

В первой строке следующей последовательности команд переменной memo присваивается строковое значение max*. Оболочка Bourne Again Shell не расширяет строку, поскольку bash не осуществляет расширение имени (см. раздел «Генерация имени файла и расширение имени» на с. 157), когда присваивает значение переменной. Все оболочки обрабатывают командную строку в определенном порядке. Согласно этому порядку, bash (но не tcsh) расширяет переменные до того, как интерпретирует команды. В следующей командной строке, вызывающей команду echo, двойные кавычки цитируют звездочку (*) в расширяемом значении переменной $memo и удерживают bash от расширения имени в расширяемой переменной memo перед передачей ее значения команде echo:

```
$ memo=max*
$ echo "$memo"
max*
```

Все оболочки интерпретируют специальные символы как специальные, когда осуществляется ссылка на переменную, содержащую непроцитированный специальный символ. В следующем примере оболочка расширяет значение переменной memo, поскольку ее имя не взято в кавычки:

```
$ ls
max.report
max.summary
$ echo $memo
max.report max.summary
```

Здесь оболочка расширяет переменную `$memo` в `max*`, расширяет `max*` в `max.report` и `max.summary` и передает эти два значения команде echo.

Фигурные скобки Синтаксис `$ПЕРЕМЕННАЯ` является частным случаем более общего синтаксиса `${ПЕРЕМЕННАЯ}`, в котором имя переменной заключено в структуру `${}`. Фигурные скобки изолируют имя переменной от примыкающих символов. Фигурные скобки необходимы, когда значение переменной соединяется со строкой:

```
$ PREF=counter
$ WAY=$PREFclockwise
$ FAKE=$PREFfeit
$ echo $WAY $FAKE

$
```

Предыдущий пример не работает так, как было задумано. При выводе отображается только пустая строка. Так происходит потому, что хотя обозначения `PREFclockwise` и `PREFfeit` являются допустимыми именами, значения этих переменных не установлены. По умолчанию оболочка вычисляет переменные с неустановленными значениями как пустые строки и выводит это значение (bash) или генерирует сообщение об ошибке (tcsh). Чтобы достичь задуманного в этих инструкциях, на переменную `PREF` следует ссылаться с использованием фигурных скобок:

```
$ PREF=counter
$ WAY=${PREF}clockwise
$ FAKE=${PREF}feit
$ echo $WAY $FAKE
counterclockwise counterfeit
```

Оболочка Bourne Again Shell ссылается на аргументы в своей командной строке по их позиции, используя специальные переменные `$1`, `$2`, `$3` и так далее, вплоть до `$9`. Если нужно сослаться на аргумент, следующий за девятым, необходимо воспользоваться фигурными скобками: `${10}`. Имя команды содержится в переменной `$0` (см. раздел «$0: Имя вызванной программы» на с. 459).

unset: Удаление переменной

Переменная не будет удалена, она продолжает свое существование, пока существует оболочка, в которой она была создана. Для удаления значения переменной, но не самой переменной этой переменной следует присвоить пустое значение (в tcsh следует использовать `set person =`):

```
$ person=
$ echo $person
$
```

Удалить переменную можно с помощью встроенной команды unset. Следующая команда удаляет переменную `person`:

```
$ unset person
```

Атрибуты переменных

В этом разделе рассматриваются атрибуты переменных и объясняется, как они устанавливаются.

readonly: Превращение значение переменной в постоянную величину

Чтобы гарантировать невозможность изменения значения переменной, можно воспользоваться встроенной командой readonly (но не в tcsh). В следующем примере person объявляется переменной только для чтения. Перед объявлением переменной только для чтения ей нужно присвоить значение, и после этого объявления ее значение изменить уже невозможно. При попытке удаления или изменения значения переменной, объявленной только для чтения, оболочка выводит сообщение об ошибке:

```
$ person=zach
$ echo $person
zach
$ readonly person
$ person=helen
bash: person: readonly variable  person: переменная только для чтения
```

Если встроенная команда readonly используется без аргумента, она выводит список всех переменных оболочки, которые могут использоваться только для чтения. Этот список включает переменные, являющиеся ключевыми словами, которые автоматически вводятся в режим только для чтения, наряду с теми переменными, являющимися ключевыми словами, или пользовательскими переменными, которые были объявлены вами только для чтения. См. пример в разделе «Вывод списка атрибутов переменных» (такую же выходную информацию можно получить с помощью команд readonly и declare -r на с. ???).

declare и typeset: Установка атрибутов переменным

Встроенные команды declare (объявление) и typeset (установка типа) (два имени для одной и той же команды, ни одна из которых не доступна в tcsh) устанавливают атрибуты и значения для переменных оболочки. В табл. 8.3 приводится список пяти атрибутов.

Таблица 8.3. Атрибуты переменных (для typeset или declare)

Атрибут	Значение
-a	Объявляет переменную типа массив (см. раздел «Переменная типа массив» на с. ???)
-f	Объявляет переменную типа имя функции (см. раздел «Функции» на с. 363)
-i	Объявляет целочисленную переменную (см. раздел «Целые числа» на с. 311)
-r	Делает переменную используемой только для чтения, см. также команду readonly
-x	Экспортирует переменную (превращает ее в глобальную), см. также команду export (см. раздел «export» на с. 454)

Следующие команды объявляют несколько переменных и устанавливают ряд атрибутов. В первой строке объявляется переменная person1 и ей присваивается значение

max. Эта команда имеет такое же действие, как с применением слова declare, так и без его применения.

```
$ declare person1=max
$ declare -r person2=zach
$ declare -rx person3=helen
$ declare -x person4
```

Встроенные команды readonly и export являются синонимами для команд declare -r и declare -x соответственно. Переменную можно объявить, не присваивая ей значение, что показано в предыдущем объявлении переменной person4. Это объявление делает переменную person4 доступной для всех подчиненных оболочек (то есть делает ее глобальной). Пока переменной не будет присвоено другое значение, ее значением будет пустая строка.

Ключи для команды declare можно перечислить отдельно (в любом порядке). Следующая команда является эквивалентом предыдущего объявления переменной person3:

```
$ declare -x -r person3=helen
```

Если нужно удалить атрибут из переменной, следует вместо символа – использовать символ +. Атрибут «только для чтения» удалить невозможно. После того как будет введена следующая команда, переменная person3 перестанет быть экспортируемой, но по-прежнему останется только для чтения.

```
$ declare +x person3
```

Вместо команды declare можно использовать команду typeset.

Вывод списка атрибутов переменных

Если ввести команду declare без ключей и аргументов, она выведет список всех переменных оболочки. Такой список выводится при запуске без аргументов команды set (см. раздел «set: Инициализация аргументов командной строки» на с. 461). Если встроенная команда declare используется с ключами, но без имен переменных в качестве аргументов, она выводит список всех переменных оболочки, у которых установлены указанные атрибуты. Например, команда declare -r выводит список всех переменных оболочки, предназначенных только для чтения. Этот список аналогичен тому, который выводится командой readonly, введенной без аргументов. После показанного в предыдущем примере объявления переменных будут получены следующие результаты:

```
$ declare -r
declare -ar BASH_VERSINFO='([0]="3" [1]="2" [2]="39" [3]="1" … )'
declare -ir EUID="500"
declare -ir PPID="936"
declare -r SHELLOPTS="braceexpand:emacs:hashall:histexpand:history:…"
declare -ir UID="500"
declare -r person2="zach"
declare -rx person3="helen"
```

Первые пять записей относятся к переменным, являющимся ключевыми словами, которые объявлены только для чтения в автоматическом режиме. Некоторые из этих переменных сохранены как целые числа (-i). Ключ -a показывает, что переменная BASH_VERSINFO относится к массивам; при этом значение каждого элемента массива показано в списке справа от знака равенства.

Целые числа

По умолчанию значения переменных хранятся в виде строк. При осуществлении арифметических операций над строковой переменной оболочка преобразует ее значение в число, обрабатывает его, а затем снова преобразует его обратно в строку. Переменные с атрибутом integer хранятся как целое число. Установка атрибута integer осуществляется следующим образом:

```
$ declare -i COUNT
```

Переменные, являющиеся ключевыми словами

Переменные, являющиеся ключевыми словами, либо наследуются, либо объявляются и инициализируются оболочкой при ее запуске. Этим переменным можно присваивать значения из командной строки или из файла запуска. Обычно необходимо, чтобы эти переменные применялись во всех запускаемых подчиненных оболочках, а также в вашей оболочке, используемой для входа в систему. Чтобы сделать переменные, которые не экспортируются оболочкой в автоматическом режиме, доступными дочерним оболочкам, нужно применить команду export (bash; см. раздел «export» на с. 454) или команду setenv (tcsh; см. раздел «Подстановка переменных» на с. 384).

Переменная HOME: Ваш личный каталог

По умолчанию личным каталогом является тот каталог, который стал рабочим после входа в систему. Ваш личный каталог создается при настройке вашей учетной записи, при работе в Linux его имя хранится в файле /etc/passwd. В Mac OS X для хранения этой информации используется Open Directory.

```
$ grep sam /etc/passwd
sam:x:501:501:Sam S. x301:/home/sam:/bin/bash
```

При входе в систему оболочка наследует имя вашего личного каталога и присваивает его переменной HOME. При вводе без аргументов команды cd рабочим становится каталог, чье имя хранится в переменной HOME:

```
$ pwd
/home/max/laptop
$ echo $HOME
/home/max
$ cd
$ pwd
/home/max
```

В данном примере показаны значение переменной HOME и результат вызова встроенной команды cd. После выполнения команды cd, не имеющей аргумента, имя рабочего каталога становится таким же, как значение переменной HOME, и представляет собой имя вашего личного каталога.

Тильда (~)

Значение переменной HOME используется оболочкой для расширения имен, в которых для обозначения личного каталога пользователя применяется сокращенная запись тильда (~) (см. раздел «Знак ~ (тильда) в полных именах» на с. 105). В следующем примере для вывода значения этой сокращенной записи используется команда echo, а затем, для вывода списка всех файлов в принадлежащем

пользователю Max каталоге laptop, который является подкаталогом личного каталога этого пользователя, используется команда ls:

```
$ echo ~
/home/max
$ ls ~/laptop
tester count lineup
```

Переменная PATH: Указание тех мест, где оболочка ищет программы

Когда в качестве команды оболочке дается не простое имя файла, а абсолютное или относительное имя, она ищет в указанном каталоге исполняемый файл с указанным именем. Если файл с указанным именем не существует, оболочка сообщает, что команда не найдена. Если файл по указанному пути существует, но у вас нет прав на его выполнение или в случае использования сценария оболочки у вас нет прав доступа на чтение и выполнение файла сценария, оболочка сообщает об отсутствии прав.

Если в качестве команды дано простое имя файла, оболочка ведет поиск в конкретных каталогах (в вашем пути поиска) той программы, которую вам нужно выполнить. Она ищет в нескольких каталогах файл, чье имя совпадает с именем команды и в отношении которого у вас есть права на выполнение (для скомпилированной программы) или на чтение и выполнение (для сценария оболочки). Управление этим поиском осуществляется с помощью переменной оболочки PATH.

Значение по умолчанию для переменной PATH определяется при компиляции программы bash или tcsh. Оно не устанавливается в файле запуска, хотя в нем оно может быть изменено. Обычно по умолчанию определяется, что оболочка ведет поиск в нескольких системных каталогах, используемых для хранения наиболее востребованных команд. В число этих системных каталогов входят /bin и /usr/bin, а также другие каталоги, имеющие отношение к локальной системе. Если при вводе команды оболочка не находит исполняемый, а в случае сценария оболочки еще и читаемый файл, чье имя совпадает с именем команды, ни в одном из каталогов, перечисленных в переменной PATH, она генерирует одно из вышеупомянутых сообщений об ошибке.

Рабочий каталог

В переменной PATH каталоги указываются в том порядке, в котором оболочка будет вести в них поиск. Каждый каталог должен быть отделен от следующего каталога двоеточием. В следующей команде устанавливается такое значение переменной PATH, чтобы поиск исполняемого файла начинался с каталога /usr/local/bin. Если в этом каталоге файл не будет найден, оболочка продолжит поиск в каталоге /bin, а затем в каталоге /usr/bin. Если поиск в этих каталогах не увенчается успехом, оболочка будет искать файл в каталоге ~/bin, который является подкаталогом личного каталога пользователя. И наконец, оболочка будет вести поиск в рабочем каталоге. Экспорт переменной PATH делает ее значение доступным для подчиненных оболочек:

```
$ export PATH=/usr/local/bin:/bin:/usr/bin:~/bin:
```

Пустая строка обозначает рабочий каталог. В предыдущем примере пустое значение (отсутствие какого-либо символа между двоеточием и концом строки) появляется в качестве последнего элемента строки. Рабочий каталог может быть представлен в виде лидирующего двоеточия (что не рекомендуется делать из соображений безопас-

ности, изложенных в следующей врезке), замыкающего двоеточия (как в данном примере) или двух следующих друг за другом двоеточий в любом месте строки. Рабочий каталог можно обозначить и в явном виде, воспользовавшись символом точки (.). Пример для оболочки tcsh показан в разделе «path или PATH» на с. 391.

Поскольку Linux хранит многие исполняемые файлы в каталогах bin (binary — двоичный), то многие пользователи помещают свои исполняемые файлы в свои собственные каталоги ~/bin. Если поместить свой каталог (как показано в предыдущем примере) в самый конец значения переменной PATH, то оболочка будет искать в нем любые команды, которые не сможет найти в каталогах, перечисленных в PATH до него.

PATH И БЕЗОПАСНОСТЬ

Из соображений безопасности рабочий каталог не следует указывать в переменной PATH первым. Если вы работаете с root-правами, указывать рабочий каталог в переменной PATH первым нельзя ни в коем случае. Для привилегированного пользователя рабочий каталог в переменной PATH обычно вообще не указывается. Файл из рабочего каталога всегда можно выполнить, поставив впереди его имени ./, например . /myprog.

Указание рабочего каталога в переменной PATH на первом месте может привести к возникновению дыры в системе безопасности. При входе в каталог многие пользователи в первую очередь вводят команду ls. Если владелец каталога поместил в него исполняемый файл с именем ls и рабочий каталог фигурирует в пользовательской переменной PATH на первом месте, то пользователь, вводящий команду ls из каталога, вызывает вместо системной утилиты ls выполнение программы ls, находящейся в рабочем каталоге, что может привести к весьма нежелательным последствиям.

Если нужно добавить к значению переменной PATH какие-нибудь каталоги, то при установке нового значения этой переменной можно сослаться на ее старое значение (но при этом следует обратить внимание на предыдущее предупреждение, касающееся безопасности). Следующая команда добавляет каталог /usr/local/bin в начало текущего значения переменной PATH и каталог bin, находящийся в личном каталоге пользователя (~/bin), в его конец:

```
$ PATH=/usr/local/bin:$PATH:~/bin
```

Переменная MAIL: Указание места хранения электронной почты

Переменная MAIL (при работе с tcsh — mail) содержит имя файла, содержащего вашу электронную почту (ваш почтовый ящик, обычно это /var/mail/*имя*, где *имя* — это ваше имя пользователя). Если установлена переменная MAIL и не установлена переменная MAILPATH (см. далее), оболочка информирует вас при поступлении электронной почты в файл, указанный в переменной MAIL. При использовании графической среды можно отключить переменную MAIL, чтобы оболочка не выводила почтовые напоминания в окно эмулятора терминала (если предположить, что вы пользуетесь графической почтовой программой).

Большинство систем Mac OS X не используют локальный файл для входящих почтовых сообщений. Вместо этого сообщения обычно содержатся на удаленном почтовом сервере. Если у вас есть локальный сервер электронной почты, то наличие переменной MAIL и других переменных оболочки, связанных с электронной почтой, не приводит к каким-либо действиям.

Переменная MAILPATH (не доступная при работе с tcsh) содержит список имен файлов, отделенных друг от друга двоеточиями. Если эта переменная установлена,

оболочка информирует вас об изменении любого из этих файлов (например, при поступлении электронной почты). После любого имени файла, присутствующего в этом списке, можно поставить вопросительный знак (?), за которым можно поставить сообщение. Пока вы будете находиться в системе, это сообщение будет заменять то почтовое сообщение, которое выводится при поступлении электронной почты.

Переменная MAILCHECK (не доступная при работе с tcsh) определяет в секундах частоту проверки наличия новых почтовых поступлений. По умолчанию устанавливается значение, равное 60 с. Если установить для этой переменной нулевое значение, оболочка будет проверять почту перед выводом каждого приглашения на ввод команды.

Переменная PS1: Приглашение пользователю (первичное)

По умолчанию в оболочке Bourne Again Shell в качестве приглашения пользователю на ввод команды используется знак доллара ($). При запуске bash с root-правами эта оболочка обычно выводит приглашение в виде знака решетки (#). Переменная PS1 (а при работе с tcsh переменная prompt; см. раздел «prompt» на с. 392) содержит строку приглашения, которую оболочка использует, чтобы оповестить вас о том, что она ожидает ввод команды. При изменении значение переменной PS1 или переменной prompt изменяется внешний вид вашего приглашения.

Приглашение, выводимое с помощью переменной PS1, можно настроить. Если, к примеру, ему присвоить значение

```
$ PS1="[\u@\h \W \!]$ "
```

будет выведено следующее приглашение:

```
[пользователь@хост каталог событие]$
```

где *пользователь* — это имя пользователя, *хост* — это имя хоста вплоть до первой точки, *каталог* — это базовое имя рабочего каталога, а *событие* — это номер события (см. раздел «Номер события» на с. 325) текущей команды.

Если вы работаете на более чем одной системе, то полезно вставить в ваше приглашение имя системы. Например, можно снабдить приглашение именем используемой системы, за которым следует двоеточие и пробел (пробел в конце приглашения упрощает чтение вводимой после приглашения команды). В этой команде используется подстановка команд (см. раздел «Командная подстановка» на с. 358) в строке, присвоенной переменной PS1:

```
$ PS1="$(hostname): "
bravo.example.com: echo test
test
bravo.example.com:
```

При работе с tcsh нужно использовать следующую команду:

```
tcsh $ set prompt = "'hostname': "
```

В первом из следующих примеров приглашение изменяется на имя локального хоста, пробел и знак доллара (или, если пользователь работает с root-правами, знак решетки). Во втором примере приглашение изменяется на показания времени, за которыми следует имя пользователя. В третьем примере приглашение изменяется на одно из используемых в данной книге (знак решетки для привилегированного и знак доллара для обычного пользователя):

```
$ PS1='\h \$ '
```

```
bravo $
$ PS1='\@ \u $ '
09:44 PM max $
$ PS1='\$ '
$
```

В табл. 8.4 приведены некоторые символы, которые можно использовать в значении переменной PS1. Соответствующие символы для настройки приглашения в оболочке tcsh приведены в табл. 9.4. Чтобы получить полный список специальных символов, пригодных для использования в строке приглашения, нужно открыть man-страницу, посвященную bash, и найти второе появление слова PROMPTING (ввести команду /PROMPTING, а затем нажать клавишу n).

Таблица 8.4. Символы, используемые в переменной PS1

Символ	Отображение в приглашении
\$	#, если пользователь работает с root-правами, в остальных случаях — $
\w	Путевое имя рабочего каталога
\W	Базовое имя рабочего каталога
\!	Номер текущего события (истории) (см. раздел «!*n* номер события» на с. 330)
\d	Дата в формате неделя, месяц, дата
\h	Имя хоста машины без указания домена
\H	Полное имя хоста машины, включая домен
\u	Пользовательское имя текущего пользователя
\@	Текущее время дня в 12-часовом формате AM/PM
\T	Текущее время дня в 12-часовом формате ЧЧ:ММ:СС
\A	Текущее время дня в 24-часовом формате ЧЧ:ММ
\t	Текущее время дня в 24-часовом формате ЧЧ:ММ:СС

Переменная PS2: Приглашение пользователю (вторичное)

В переменной PS2 содержится вторичное приглашение. (В оболочке tcsh используется переменная prompt2; см. раздел «Переменная prompt2» на с. 392.) В первой строке следующего примера за командой echo следует не закрытая в кавычках строка. Оболочка предполагает, что команда не закончена, и на второй строке выводит вторичное приглашение, используемое по умолчанию (>). Это приглашение показывает, что оболочка ожидает от пользователя продолжения ввода командной строки. Оболочка ждет до тех пор, пока не будет получена кавычка, закрывающая строку. И только после этого она выполняет команду:

```
$ echo "демонстрация строки приглашения
> 2"
Демонстрация строки приглашения
2
$ PS2="вторичное приглашение: "
$ echo "здесь демонстрируется
Вторичное приглашение: строка приглашения 2"
здесь демонстрируется
строка приглашения 2
```

Вторая команда изменяет вторичное приглашение на `вторичное приглашение:`, за которым следует пробел. Новое приглашение демонстрируется на примере многострочной команды echo.

Переменная PS3: Приглашение меню

Переменная `PS3` содержит приглашение меню (в tcsh используется переменная `prompt3`; см. раздел «prompt3» на с. 393) для управляющей структуры выбора (см. раздел «PS3» на с. 496).

Переменная PS4: Приглашение к отладке

Переменная `PS4` содержит символ отладки bash (см. раздел «Отладка сценариев оболочки» на с. 428; в оболочке tcsh аналог этой переменной отсутствует).

Переменная IFS: Разделение полей ввода (разбиение слов)

Переменная оболочки `IFS` (Internal Field Separator — внутренний разделитель полей), которая не имеет аналога в оболочке tcsh, определяет символы, которые можно использовать для разделения аргументов в командной строке. У нее есть значение по умолчанию, состоящее из пробела, символа табуляции и символа разделителя строк (SPACE TAB NEWLINE). Независимо от значения переменной `IFS`, вы всегда можете использовать для отделения друг от друга аргументов командной строки один или несколько пробелов или символов табуляции, при условии, что эти символы не процитированы или не отключены. Когда переменной `IFS` присваивается символьное значение, эти символы могут также разделять поля, но только если они подвергаются расширению. Этот тип интерпретации командной строки называется *разбиением слов*.

ИЗМЕНЯТЬ ЗНАЧЕНИЯ IFS НУЖНО ОСТОРОЖНО

Изменение значения переменной IFS имеет множество побочных эффектов, поэтому с ней нужно обращаться осторожно. Возможно, перед изменением значения переменной IFS будет полезно сохранить ее прежнее значение. Тогда при получении неожиданных результатов можно будет легко восстановить ее исходное значение. Кроме этого, можно перед экспериментированием с переменной IFS запустить новую дочернюю оболочку с командой bash; если возникнут проблемы, можно будет выйти и вернуться в родительскую оболочку, в которой настройки IFS работают должным образом.

В следующем примере демонстрируется, как установка значения переменной `IFS` может воздействовать на интерпретацию командной строки:

```
$ a=w:x:y:z

$ cat $a
cat: w:x:y:z: No such file or directory  Такой файл или каталог не существует
$ IFS=":"

$ cat $a
cat: w: No such file or directory        Такой файл или каталог не существует
cat: x: No such file or directory        Такой файл или каталог не существует
cat: y: No such file or directory        Такой файл или каталог не существует
cat: z: No such file or directory        Такой файл или каталог не существует
```

При первом вызове утилиты cat оболочка расширяет переменную a, интерпретируя строку w:x:y:z как единое слово, используемое в качестве аргумента утилиты cat. Утилита cat не может найти файл по имени w:x:y:z и сообщает об ошибке, связанной с этим именем файла. После того как значение переменной IFS установлено на двоеточие (:), оболочка расширяет переменную, превращая ее значение в четыре слова, каждое из которых является аргументом для утилиты cat. Теперь cat сообщает об ошибках, связанных с именами четырех файлов: w, x, y и z. Разбиение слов, основанное на двоеточии (:), осуществляется только *после* расширения переменной.

Оболочка разбивает все *расширенные* слова в командной строке в соответствии с символами разделения, найденными в переменной IFS. Если расширения не происходит, то нет и разбиения. Рассмотрим следующие команды:

```
$ IFS="p"
$ export VAR
```

Хотя переменной IFS присвоено значение p, буква p в командной строке export не расширяется, поэтому слово *export* не разбивается.

В следующем примере используется расширение переменной в попытке получить команду export:

```
$ IFS="p"
$ aa=export
$ echo $aa
ex ort
```

На этот раз происходит расширение, поэтому символ p в лексеме export интерпретируется как разделитель (что показывает команда echo). Теперь при попытке использования значения переменной aa для экспорта переменной VAR оболочка производит синтаксический разбор командной строки $aa VAR как ex ort VAR. В результате командная строка запускает редактор ex с двумя именами файлов: ort и VAR.

```
$ $aa VAR
2 files to edit          редактирование двух файлов
"ort" [New File]         "ort" [Новый Файл]
```

Войдите в режим Ex. Наберите «visual», чтобы перейти в нормальный режим.

```
:q
E173: 1 more file to edit    Есть еще один файл для редактирования
:q
$
```

Если переменной IFS вернуть прежнее значение, в качестве разделителей полей будут работать только пробелы и символы табуляции.

НЕСКОЛЬКО СИМВОЛОВ-РАЗДЕЛИТЕЛЕЙ

Хотя оболочка интерпретирует последовательность из нескольких пробелов или символов табуляции в качестве одного разделителя, она рассматривает каждое появление другого символа-разделителя полей в качестве отдельного разделителя.

Переменная CDPATH: Расширение возможностей команды cd

Переменная CDPATH (в tcsh переменная cdpath) позволяет использовать простое имя файла в качестве аргумента для встроенной команды cd для смены рабочего каталога на каталог, отличающийся от дочернего каталога рабочего каталога. Если есть

несколько каталогов, с которыми обычно проводится работа, эта переменная может ускорить работу и сэкономить усилия при использовании утилиты cd с длинными именами для переключения между каталогами, на которые они указывают.

Когда значение для переменной CDPATH или cdpath не установлено и в качестве аргумента утилите cd указывается простое имя файла, cd ищет в качестве нового рабочего каталога подкаталог с именем, указанным в аргументе. Если подкаталог с таким именем не существует, cd выводит сообщение об ошибке. Когда значение для переменной CDPATH или cdpath установлено, cd ищет подходящее имя подкаталога в каталогах из списка, указанного в CDPATH. Если подкаталог с таким именем будет найден, он становится рабочим каталогом. После установки значения переменной CDPATH или cdpath можно использовать cd и простое имя файла для смены рабочего каталога на дочерний подкаталог или на любые каталоги, перечисленные в переменной CDPATH или cdpath.

Переменная CDPATH или cdpath принимает значение, представляющее собой список имен каталогов, разделенных двоеточием (наподобие значения переменной PATH). Ее значение обычно устанавливается в файле загрузки ~/.bash_profile (bash) или ~/.tcshrc (tcsh) с помощью командной строки, имеющей следующий вид:

```
export CDPATH=$HOME:$HOME/literature
```

Для tcsh нужно использовать следующий формат:

```
setenv cdpath $HOME\:$HOME/literature
```

Эти команды заставляют утилиту cd при вводе команды cd вести поиск в вашем личном каталоге, в каталоге literature, а затем в рабочем каталоге. Если в значение переменной CDPATH или cdpath не включить рабочий каталог, то утилита cd будет вести поиск в рабочем каталоге только в том случае, когда не удастся найти подкаталог с указанным именем во всех других каталогах, перечисленных в значении переменной CDPATH или cdpath. Если нужно, чтобы сначала команда cd вела поиск в рабочем каталоге, нужно в качестве первой записи включить в значение переменной CDPATH пустую строку, представленную двумя двоеточиями (::):

```
export CDPATH=::$HOME:$HOME/literature
```

Если аргумент для встроенной команды cd содержит абсолютный путь, то есть путь, который начинается с символа слэша (/), оболочка не будет обращаться к значению переменной CDPATH или cdpath.

Переменные, являющиеся ключевыми словами: заключение

В табл. 8.5 представлен список переменных bash, являющихся ключевыми словами. Сведения об аналогичных переменных оболочки tcsh даны в разделе «Переменные оболочки» на с. 390.

Таблица 8.5. Переменные bash, являющиеся ключевыми словами

Переменная	Значение
BASH_ENV	Путевое имя файла загрузки для неинтерактивных оболочек (раздел «Неинтерактивные оболочки» на с. 285)
CDPATH	Путь поиска для утилиты cd (раздел «Переменная CDPATH: Расширение возможностей команды cd» на с. 317)
COLUMNS	Ширина отображения, используемая структурой управления select (раздел «Управляющая структура select» на с. 445)

Переменная	Значение
FCEDIT	Имя редактора, используемого встроенной командой fc по умолчанию (раздел «Команда fc: Отображение, редактирование и повторное выполнение команд» на с. 327)
HISTFILE	Путевое имя файла, в котором хранится список предыстории (по умолчанию используется файл ~/.bash_history; см. раздел «История» на с. 324)
HISTFILESIZE	Максимальное количество записей, хранящихся в файле, указанном в HISTFILE (по умолчанию 500; см. раздел «Переменные, управляющие историей» на с. 324)
HISTSIZE	Максимальное количество записей, хранящихся в списке предыстории (по умолчанию 500; см. раздел «История» на с. 324)
HOME	Путевое имя личного каталога пользователя (раздел «Переменная HOME: Ваш личный каталог» на с. 311); используется в качестве аргумента по умолчанию для команды cd и в расширении тильды (раздел «Знак ~ (тильда) в полных именах» на с. 104)
IFS	Внутренний разделитель полей (раздел «Переменная IFS: Разделение полей ввода (разбиение слов)» на с. 316); используется для разбиения слов (раздел «Разбиение слов» на с. 359)
INPUTRC	Путевое имя файла запуска библиотеки Readline (по умолчанию используется файл ~/.inputrc; см. раздел «Файл .inputrc: Конфигурирование библиотеки Readline» на с. 338)
LANG	Категория локализации, когда эта категория специально не указана с помощью переменной LC_*
LC_*	Группа переменных, определяющая категории локализации, включающая LC_COLLATE, LC_CTYPE, LC_MESSAGES и LC_NUMERIC; использует встроенную команду locale для вывода полного списка значений
LINES	Высота отображения, используемая структурой управления select (раздел «Управляющая структура select» на с. 445)
MAIL	Путевое имя файла, в котором хранится электронная почта пользователя (раздел «Переменная MAIL: Указание места хранения электронной почты» на с. 313)
MAILCHECK	Частота проверки электронной почты оболочкой bash, заданная в секундах (раздел «Переменная MAIL: Указание места хранения электронной почты» на с. 313)
MAILPATH	Список с двоеточием в качестве разделителя, который оболочка bash использует для проверки на наличие поступившей почты (раздел «Переменная MAIL: Указание места хранения электронной почты» на с. 313)
PATH	Список имен каталогов с двоеточием в качестве разделителя, который используется bash для поиска в этих каталогах команд (раздел «Переменная PATH: Указание тех мест, где оболочка ищет программы» на с. 312)
PROMPT_COMMAND	Команда, которую bash выполняет непосредственно перед выводом первичного приглашения
PS1	Строка приглашения 1; первичное приглашение (раздел «Переменная PS1: Приглашение пользователю (первичное)» на с. 314)
PS2	Строка приглашения 2; вторичное приглашение (по умолчанию «>»; см. раздел «Переменная PS2: Приглашение пользователю (вторичное)» на с. 315)
PS3	Приглашение, выводимое структурой управления select (раздел «Управляющая структура select» на с. 445)

продолжение ⇩

Таблица 8.5 (*продолжение*)

Переменная	Значение
PS4	Символ отладки bash (раздел «Отладка сценариев оболочки» на с. 428)
REPLY	Содержит строку, которая воспринимается встроенной командой read (раздел «REPLY» на с. 467); также используется структурой управления select (раздел «Управляющая структура select» на с. 445)

Специальные символы

В табл. 8.6 перечислено большинство символов, имеющих специальное значение для оболочек bash и tcsh.

Таблица 8.6. Специальные символы оболочки

Символ	Использование
NEWLINE (разделитель строк)	Инициирование выполнения команды (раздел «Точка с запятой (;) и перевод строки (↵) разделяют команды» на с. 296)
;	Разделение команды (раздел «Точка с запятой (;) и перевод строки (↵) разделяют команды» на с. 296)
( )	Группировка команд (раздел «средство группировки команд» на с. 298) для выполнения подчиненной оболочкой или идентификация функции (раздел «Функции» на с. 369)
(( ))	Расширение арифметического выражения (раздел «Арифметическое расширение» на с. 356)
&	Выполнение команды в фоновом режиме (раздел «Запуск команды в фоновом режиме», а также раздел «| и & — разделяют команды, но не только» на с. 297)
|	Отправка стандартного вывода предыдущей команды на стандартный ввод следующей команды (канал; раздел «| и & — разделяют команды, но не только» на с. 297)
>	Перенаправление стандартного вывода (раздел «Перенаправление стандартного вывода» на с. 146)
>>	Добавление стандартного вывода (раздел «Добавление стандартного вывода в файл» на с. 150)
<	Перенаправление стандартного ввода (раздел «Перенаправление стандартного ввода» на с. 147)
<<	Начало структуры Here document (раздел «Структура Here document» на с. 442)
*	Любая строка из нуля или большего количества символов в нестрогой ссылке на файлы (раздел «Специальный символ „*“» на с. 158)
?	Любой одиночный символ в нестрогой ссылке на файлы (раздел «Специальный символ „?“» на с. 157)
\	Цитирование следующего символа (раздел «Обратный слэш» на с. 66)
‘	Заключение строки в кавычки предотвращает все подстановки (раздел «Одинарные кавычки» на с. 66)
“	Заключение строки в кавычки, позволяет осуществлять подстановку только переменных и команд (раздел «Специальные символы» на с. 65 и раздел «Цитирование символа $» на с. 306)

Символ	Использование
`...`	Осуществление командной подстановки (раздел «Командная подстановка» на с. 358)
[]	Класс символов в нестрогой ссылке на файлы (раздел «Специальные символы [ ]» на с. 160)
$	Ссылка на переменную (раздел «Параметры и переменные» на с. 304)
. (встроенная команда точка)	Выполнение команды (раздел «Утилита . (точка) или source: Запуск файла запуска в текущей оболочке» на с. 287)
#	Начало комментария («# — символ начала комментария» на с. 295)
{}	Окружение содержимого функции (раздел «Функции» на с. 363)
: (встроенная команда null)	Возвращение значения true (раздел «Встроенная команда : (нулевая строка)» на с. 464)
&& (булево И)	Выполнение команды справа только если команда слева была выполнена удачно (вернула статус выхода нуль; см. раздел «Булевы операции» на с. 486)
|| (булево ИЛИ)	Выполнение команды справа только если команда слева была выполнена неудачно (вернула ненулевой статус выхода; раздел «Булевы операции» на с. 486)
! (булево НЕ)	Изменение статуса выхода из команды на противоположный
$() (не в tcsh)	Осуществление командной подстановки (предпочтительная форма; см. раздел «Командная подстановка» на с. 358)
[]	Вычисление арифметического выражения (раздел «Арифметическое расширение» на с. 356)

Процессы

Процессом называется выполнение команды ядром Linux. Оболочка, которая была запущена при входе в систему, является командой или процессом, как и любая другая команда. Когда в командную строку вводится имя утилиты Linux, инициируется процесс. Когда запускается сценарий оболочки, начинается еще один процесс оболочки, и дополнительные процессы создаются для каждой, имеющейся в сценарии команды. В зависимости от способа вызова сценария оболочки сценарий работает под управлением либо текущей оболочки, либо, что более типично, под управлением подчиненной текущей оболочки дочерней оболочки. При запуске встроенной команды оболочки, например cd, процесс не запускается.

Структура процессов

Системный вызов fork

Структура процесса имеет иерархический характер, похожий на файловую структуру, со своими родительскими, дочерними и даже *корневыми* элементами. Родительский процесс *ответвляет* от себя дочерний процесс, который, в свою очередь, может ответвить от себя другие процессы. (Термин *ответвление* (fork) свидетельствует о том, что, подобно дорожной развилке, один процесс превращается в два. Сначала оба разветвления идентичны друг другу, за исключением того, что один идентифицируется как родительский, а второй — как дочерний. Часто также используется термин *порожденный* (spawn); эти два слова являются взаимозаменяемыми.) Подпрограмма операционной системы, или *системный вызов*, которая создает новый процесс, называется fork().

Когда при запуске системы начинает работать операционная система Linux, она запускает init, одиночный процесс, который называется *спонтанным процессом*, и имеет PID-номер 1. Этот процесс занимает в структуре процессов такое же место, что и корневой каталог в файловой структуре: он является предком всех процессов, с которыми работает система и пользователи. Когда система командной строки находится в многопользовательском режиме, init запускает процессы getty или mingetty, которые отображают на терминалах приглашение на вход в систему `login:`. Когда пользователь отвечает на приглашение и нажимает клавишу ВВОД, getty передает управление утилите по имени login, которая проверяет комбинацию из имени пользователя и пароля. После входа пользователя в систему процесс login становится процессом пользовательской оболочки.

Идентификация процесса

PID-номера В начале работы каждого процесса Linux присваивает ему PID-номер (process identification — идентификация процесса). Пока процесс существует, за ним сохраняется один и тот же PID-номер. В течение одного сеанса оболочка входа в систему выполняется в одном и том же процессе. Когда ответвляется новый процесс, например при использовании редактора, PID-номер нового (дочернего) процесса отличается от номера своего родительского процесса. При возврате в оболочку входа в систему она по-прежнему выполняется тем же самым процессом и имеет тот же самый PID-номер, присвоенный этому процессу при входе в систему.

В следующем примере показано ответвление от процесса, в котором работает оболочка (родительского процесса), процесса, запускающего утилиту ps. При вызове с ключом `-f` утилита ps выводит полный список сведений о каждом процессе. Строка ps, в которой в столбце CMD выведена запись `bash`, ссылается на процесс, запустивший оболочку. В столбце, озаглавленном PID, показываются PID-номера. В столбце, озаглавленном PPID, показывается PID-номер *родительского* процесса. На основании сведений в столбцах PID и PPID можно прийти к выводу, что процесс, в котором запущена оболочка (PID 21341), является родительским по отношению к процессу, запустившему утилиту sleep (PID 22789).

Родительский PID-номер sleep такой же, как PID-номер оболочки (21341).

```
$ sleep 10 &
[1] 22789
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
max      21341 21340  0 10:42 pts/16   00:00:00 bash
max      22789 21341  0 17:30 pts/16   00:00:00 sleep 10
max      22790 21341  0 17:30 pts/16   00:00:00 ps -f
```

Дополнительные сведения об утилите ps и о столбцах, которые она выводит при вызове с ключом `-f`, даны в части V. Вторая пара команд `sleep` и `ps -f` показывает, что оболочка по-прежнему запущена в том же самом процессе, но от него ответвлен новый процесс для запуска утилиты sleep:

```
$ sleep 10 &
[1] 22791
$ ps -f
```

```
UID        PID  PPID  C STIME TTY          TIME CMD
max      21341 21340  0 10:42 pts/16   00:00:00 bash
max      22791 21341  0 17:31 pts/16   00:00:00 sleep 10
max      22792 21341  0 17:31 pts/16   00:00:00 ps -f
```

Чтобы посмотреть на взаимоотношения между родительскими и дочерними процессами, можно также воспользоваться командой **pstree** (или ввести команду `ps --forest` с ключом или без ключа `-e`). В следующем примере показан вызов утилиты **pstree** с ключом `-p`, который заставляет эту утилиту показывать PID-номера:

```
$ pstree -p
init(1)-+-acpid(1395)
        |-atd(1758)
        |-crond(1702)
        …
        |-kdeinit(2223)-+-firefox(8914)---run-mozilla.sh(8920)---firefox-bin(8925)
        |               |-gaim(2306)
        |               |-gqview(14062)
        |               |-kdeinit(2228)
        |               |-kdeinit(2294)
        |               |-kdeinit(2314)-+-bash(2329)---ssh(2561)
        |               |               |-bash(2339)
        |               |               '-bash(15821)---bash(16778)
        |               |-kdeinit(16448)
        |               |-kdeinit(20888)
        |               |-oclock(2317)
        |               '-pam-panel-icon(2305)---pam_timestamp_c(2307)
        …
        |-login(1823)---bash(20986)-+-pstree(21028)
        |                           '-sleep(21026)
        …
```

Предшествующая выходная информация представлена в сокращенном виде. Строка, начинающаяся с `-kdeinit`, показывает, что пользователь, работающий в графическом режиме, запустил множество процессов, включая `firefox`, `gaim` и `oclock`. Строка, начинающаяся с `-login`, показывает, что пользователь, работающий в текстовом режиме, запустил в фоновом режиме училиту **sleep** и в активном режиме утилиту **pstree**. Сведения о том, как заставить оболочку выводить PID-номера, даны в главе 10, в разделе «$$: PID-номер» на с. 457.

Выполнение команды

Вызовы fork и sleep

Когда оболочке дается команда, она, как правило, ответвляет от себя (порождает, используя системный вызов **fork()**) дочерний процесс, предназначенный для выполнения команды. Пока дочерний процесс выполняет команды, родительский процесс приостанавливается (это реализуется с помощью системного вызова **sleep()**). Приостановленный процесс не тратит компьютерного времени и остается неактивным, ожидая возобновления работы (пробуждения). Когда дочерний процесс завершает выполнение команды, он сообщает через статус выхода своему родительскому процессу о своем завершении (удаче или неудаче), а затем

прекращает существование. Родительский процесс (в котором запущена оболочка) пробуждается и выводит приглашение на ввод следующей команды.

Фоновый процесс Когда процесс с помощью добавления в конец команды символа амперсанда (&) запускается в фоновом режиме, оболочка ответвляет от себя дочерний процесс, но не приостанавливается и не ждет, пока дочерний процесс завершит свою работу. В таком режиме родительский процесс, в котором выполняется программа оболочки, выводит номер задания и PID-номер дочернего процесса и приглашение на ввод следующей команды. Дочерний процесс запускается в фоновом режиме, независимо от своего родителя.

Встроенные команды Хотя для большинства введенных команд оболочка ответвляет процесс для их запуска, некоторые команды встроены в оболочку. Для запуска встроенных команд оболочке не нужно ответвлять процесс. Дополнительные сведения даны в разделе «Встроенные команды» на с. 161.

Переменные Внутри отдельно взятого процесса, например того, в котором работает оболочка входа в систему или подчиненная оболочка, можно объявлять, инициализировать, читать и изменять значения переменных. Но по умолчанию переменная является локальной по отношению к процессу. Когда от процесса ответвляется дочерний процесс, родительский процесс не передает значение переменной своему дочернему процессу. Значение переменной можно сделать доступным дочернему процессу (глобальным) с помощью встроенной команды export при работе с bash (см. раздел «Команда export» на с. 454) или встроенной команды setenv при работе с tcsh (см. раздел «Строковые переменные» на с. 384).

История

Механизм истории, позаимствованный у оболочки C Shell, ведет список самых последних введенных командных строк, также называемых событиями, предоставляющий быстрый способ повторного выполнения любого события, присутствующего в списке. Этот механизм также позволяет выполнять вариации предыдущих команд и повторно использовать имеющиеся в них аргументы. Исторический список можно использовать для копирования ранее использовавшихся сложных команд и аргументов и ввода серии команд, незначительно отличающихся друг от друга. Он служит также в качестве записи всего сделанного. Он может пригодиться в том случае, если допущена ошибка и нет уверенности в том, что сделано, или когда нужно сохранить запись процедуры, вызывающей серию команд.

Встроенная команда history (в обеих оболочках, и в bash, и в tcsh) выводит исторический список. Если этого не происходит, нужно изучить следующий раздел, в котором рассматриваются переменные, требующие установки.

Переменные, управляющие историей

Механизм истории, имеющийся в оболочке TC Shell, похож на механизм, имеющийся в оболочке bash, но использует другие переменные и имеет ряд других отличий. Дополнительные сведения даны в разделе «История» на с. 372.

Значение переменной HISTSIZE определяет количество событий, сохраняемых в историческом списке на период сеанса работы. Обычно для нее устанавливается значение от 100 до 1000.

При выходе из оболочки самые последние выполненные команды сохраняются в файле, чье имя сохранено в переменной HISTFILE (по умолчанию используется имя файла ~/.bash_history). При следующем запуске оболочки исторический список инициализируется из этого файла. Значение переменной HISTFILESIZE определяет количество строк истории, сохраняемое в файле, указанном в HISTFILE (табл. 8.7).

ИСТОРИЯ МОЖЕТ ПОМОЧЬ В ОТСЛЕЖИВАНИИ ОШИБОК

Когда в командной строке допущена ошибка (за исключением ошибок в сценарии или программе), и вы не понимаете, что сделано неправильно, загляните в исторический список и пересмотрите все свои последние команды. Иногда этот список может помочь выяснить, что было не так и как все исправить.

Таблица 8.7. Переменные истории

Переменная	Значение по умолчанию	Функция
HISTSIZE	500 событий	Максимальное количество событий, сохраняемых за сеанс
HISTFILE	~/.bash_history	Размещение файла истории
HISTFILESIZE	500 событий	Максимальное количество событий, сохраняемое между сеансами

Номер события

Оболочка Bourne Again Shell присваивает последовательные номера событий каждой командной строке. Номер события можно вывести на экран как часть приглашения bash, включив в значение переменной PS1 строку \! (см. раздел «Переменная PS1: Приглашение пользователю (первичное)» на с. 314). Пронумерованные приглашения показаны в примерах данного раздела в тех случаях, когда они помогают проиллюстрировать поведение команды.

Введите следующую команду в командную строку или поместите ее в файл ~/.bash_profile, чтобы ее действие проявлялось в будущих сеансах работы, чтобы инициировать ведение исторического списка, содержащего 100 самых последних событий:

```
$ HISTSIZE=100
```

Следующая команда заставляет bash сохранить 100 самых последних событий между сеансами входа в систему:

```
$ HISTFILESIZE=100
```

После установки значения переменной HISTFILESIZE можно выйти из системы и войти в нее снова, и в вашем историческом списке появятся 100 самых последних событий из прежнего сеанса входа в систему.

Чтобы вывести исторический список, нужно ввести команду history. В этом списке сначала следуют самые старые события. Исторический список оболочки tcsh включает время выполнения команды. Следующий исторический список включает команду изменения формы приглашения, поэтому в нем отображается номер исторического события. Последней в историческом списке стоит команда history, которая вывела этот список.

```
32 $ history | tail
   23  PS1="\! bash$ "
   24  ls -l
   25  cat temp
   26  rm temp
   27  vim memo
   28  lpr memo
   29  vim memo
   30  lpr memo
   31  rm memo
   32  history | tail
```

По мере ввода команд и удлинения исторического списка он при использовании встроенной команды history может выйти за пределы верхней части экрана. Нужно перенаправлять вывод через канал утилите less и просматривать информацию через нее или же вводить команду `history 10` (или `history | tail`) для просмотра десяти *последних команд*.

УДОБНЫЙ ПСЕВДОНИМ ДЛЯ HISTORY

Создание следующих псевдонимов упрощает работу с историей. Первый псевдоним позволяет ввести команду h для вывода 10 самых последних событий. Второй псевдоним заставляет команду hg строка вывести все события в историческом списке, в которых содержится строка string. Чтобы сделать эти псевдонимы доступными после каждого входа в систему, их определения нужно поместить в ваш файл ~/.bashrc. Дополнительные сведения даны в разделе «Псевдонимы» на с. 363.

```
$ alias 'h=history | tail'
$ alias 'hg=history | grep'
```

Повторное выполнение и редактирование команд

Любое событие в историческом списке можно повторить. Эта возможность сэкономит ваше время, силы и нервы. Избавление от потребности ввода длинных командных строк позволяет упростить повторение событий, ускорить и избавить от ошибок, которые можно было бы допустить при повторном наборе всей командной строки. Ранее выполненные команды можно повторно вызывать, изменять и заново выполнять тремя способами: можно воспользоваться встроенной командой fc (рассматриваемой далее), командами восклицательных знаков (см. раздел «Использование восклицательного знака (!) для ссылки на событие» на с. 329) или библиотекой Readline, которая использует однострочный редактор, похожий на vi или emacs для редактирования и выполнения событий (см. раздел «Библиотека Readline»).

КАКИМ СПОСОБОМ ЛУЧШЕ ВОСПОЛЬЗОВАТЬСЯ?

Если вы лучше знаете vi или emacs и меньше знакомы с оболочкой C Shell или оболочкой TC Shell, пользуйтесь fc или библиотекой Readline. Если вы лучше знаете оболочку C Shell или оболочку TC Shell, пользуйтесь командами восклицательных знаков. Если вы затрудняетесь ответить на вопрос, что именно вам больше знакомо, попробуйте воспользоваться библиотекой Readline, поскольку она поможет лучше разобраться с другими областями Linux, чем изучение команд восклицательного знака.

Команда fc: Отображение, редактирование и повторное выполнение команд

Встроенная команда fc (fix command — правка команды) (ее аналог в tcsh отсутствует) позволяет выводить исторический список, а также редактировать и заново выполнять предыдущие команды. Она предоставляет многие из тех же самых возможностей, что и редакторы командной строки.

Просмотр исторического списка

При вызове команды fc с ключом `-l` она выводит команды из исторического списка. Если вызвать ее без аргументов, команда `fc -l` выводит из исторического списка 16 самых последних команд, при этом первой появляется самая старая команда:

```
$ fc -l
1024    cd
1025    view calendar
1026    vim letter.adams01
1027    aspell -c letter.adams01
1028    vim letter.adams01
1029    lpr letter.adams01
1030    cd ../memos
1031    ls
1032    rm *0405
1033    fc -l
1034    cd
1035    whereis aspell
1036    man aspell
1037    cd /usr/share/doc/*aspell*
1038    pwd
1039    ls
1040    ls man-html
```

Встроенная команда fc может воспринимать с ключом `-l` нуль, один или два аргумента. Аргументы определяют ту часть истории, которая будет выведена:

`fc -l [`*первое* `[`*последнее*`]]`

Встроенная команда fc выводит список команд, начиная с самого недавнего события, соответствующего аргументу *первое*. Аргумент может быть номером события, первыми несколькими символами командной строки или отрицательным числом, под которым подразумевается *n*-я предыдущая команда. Без аргумента *последнее* команда fc выводит события до самого последнего. Если включить в командную строку аргумент *последнее*, команда fc выводит команды от самого последнего события, соответствующего аргументу *первое*, и до самого последнего события, соответствующего аргументу *последнее*.

Следующая команда выводит исторический список от события 1030 до события 1035:

```
$ fc -l 1030 1035
1030    cd ../memos
1031    ls
1032    rm *0405
1033    fc -l
```

```
1034     cd
1035     whereis aspell
```

Следующая команда выводит самое последнее событие, которое начинается со слова view, и продолжает вывод списка событий вплоть до самой последней командной строки, которая начинается со слова whereis:

```
$ fc -l view whereis
1025     view calendar
1026     vim letter.adams01
1027     aspell -c letter.adams01
1028     vim letter.adams01
1029     lpr letter.adams01
1030     cd ../memos
1031     ls
1032     rm *0405
1033     fc -l
1034     cd
1035     whereis aspell
```

Чтобы вывести из исторического списка одну команду, для первого и второго аргументов используется один и тот же идентификатор. Следующая команда выводит событие 1027:

```
$ fc -l 1027 1027
1027     aspell -c letter.adams01
```

Редактирование и повторное выполнение предыдущих команд

Команду fc можно использовать для редактирования и повторного выполнения предыдущих команд.

fc [-e редактор] [первое [последнее]]

При вызове fc с ключом -e, за которым следует имя редактора, fc вызывает редактор с событием или событиями в Рабочем буфере, предполагая, что указанный редактор уже установлен. По умолчанию команда fc вызывает редактор nano. Без аргументов *первое* и *последнее* используется настройка по умолчанию на самую последнюю команду. В следующем примере вызывается редактор vim для редактирования самой последней команды:

```
$ fc -e vi
```

Встроенная команда fc использует автономный редактор vim. Если установлено значение переменной FCEDIT, то для указания в командной строке редактора ключ -e не нужен. Поскольку значение переменной FCEDIT было изменено на /usr/bin/emacs и fc не имеет аргументов, следующая команда редактирует последнюю команду с помощью редактора emacs (являющийся частью пакета emacs, который не устанавливается по умолчанию):

```
$ export FCEDIT=/usr/bin/emacs
$ fc
```

Если команду вызвать с одним аргументом, fc вызовет редактор для указанной команды. В следующем примере редактор запускается с событием 1029 в рабочем буфере. При выходе из редактора оболочка выполняет команду:

```
$ fc 1029
```

Как уже ранее упоминалось, команду можно идентифицировать с помощью номеров или указывая первые несколько символов командной строки. В следующем примере редактор вызывается для работы с событием из числа самых последних, которое начинается букв vim, и со всеми событиями вплоть до 1030:

```
$ fc vim 1030
```

ОЧИСТКА БУФЕРА FC

При выполнении команды fc оболочка выполняет все, что оставлено в буфере редактора, возможно, с нежелательными результатами. Если решено не выполнять команду, то перед выходом из редактора нужно очистить буфер.

Повторное выполнение команд без вызова редактора

Предыдущие команды можно выполнить повторно и без использования редактора. Если команду fc ввести с ключом -s, она пропустит фазу редактирования и выполнит команду еще раз. В следующем примере повторно происходит событие 1029:

```
$ fc -s 1029
lpr letter.adams01
```

В следующем примере снова выполняется предыдущая команда:

```
$ fc -s
```

При повторном выполнении команды можно заставить fc подставить одну строку вместо другой. В следующем примере в событии 1029 вместо строки adams подставляется строка john, и происходит измененное событие:

```
$ fc -s adams=john 1029
lpr letter.john01
```

Использование восклицательного знака (!) для ссылки на событие

Механизм истории оболочки C Shell использует для ссылки на события восклицательный знак. Использование этой технологии, которая доступна в bash и в tcsh, зачастую выглядит более громоздко, по сравнению с fc, но тем не менее у нее есть ряд полезных свойств. Например, благодаря команде !! повторно происходит предыдущее событие, и оболочка заменяет лексему !$ последним словом в предыдущей командной строке.

На событие можно сослаться, воспользовавшись его абсолютным номером события, его относительным номером события или содержащимся в нем текстом. Все ссылки на события, называемые указателями событий, начинаются с восклицательного знака (!). Для указания на событие за восклицательным знаком следуют один или несколько символов.

Историческое событие можно поместить в любом месте командной строки. Для отключения восклицательного знака, чтобы оболочка воспринимала его буквально, а не как начало исторического события, нужно поставить перед ним символ обратного слэша (\) или взять его в одинарные кавычки.

Указатели событий

Указатель события определяет команду в историческом списке. Список указателей событий приведен в табл. 8.8.

!! повторное наступление предыдущего события

С помощью команды !! можно заставить предыдущее событие наступить еще раз. В следующем примере событие 45 заставляет наступить еще раз событие 44:

```
44 $ ls -l text
-rw-rw-r--   1 max group 45 Apr 30 14:53 text
45 $ !!
ls -l text
-rw-rw-r--   1 max group 45 Apr 30 14:53 text
```

Команда !! работает независимо от того, выводится или нет в вашем приглашении номер события. Как показано в данном примере, при использовании механизма истории для инициирования повторного наступления события оболочка показывает повторно выполняемую команду.

!n номер события

Номер, следующий за восклицательным знаком, ссылается на событие. Если это событие присутствует в историческом списке, оболочка вызывает его повторное наступление. В противном случае оболочка выводит сообщение об ошибке. Отрицательный номер, следующий за восклицательным знаком, ссылается на событие относительно текущего события. Например, команда !-3 ссылается на третье по счету предыдущее событие.

После ввода команды относительный номер события, фигурирующий в заданном событии, изменяется (событие -3 становится событием -4). Обе следующие команды вызывают повторное наступление события 44:

```
51 $ !44
ls -l text
-rw-rw-r--   1 max group 45 Apr 30 14:53 text
52 $ !-8
ls -l text
-rw-rw-r--   1 max group 45 Apr 30 14:53 text
```

!строка текст события

Когда за восклицательным знаком следует строка текста, оболочка ищет и вызывает наступление самого последнего события, которое *начинается* с указанной строки. Если заключить строку в вопросительные знаки, оболочка вызывает повторное наступление самого последнего события, в котором *содержится* эта строка. Закрывающий вопросительный знак необязателен, если ВВОД нажимается сразу за введенной командой.

```
68 $ history 10
   59  ls -l text*
   60  tail text5
   61  cat text1 text5 > letter
   62  vim letter
   63  cat letter
   64  cat memo
   65  lpr memo
   66  pine zach
   67  ls -l
   68  history
69 $ !l
ls -l
```

```
...
70 $ !lpr
lpr memo
71 $ !?letter?
cat letter
...
```

Таблица 8.8. Указатели событий

Указатель	Значение
!	Начинает историческое событие, если сразу за ним следует пробел, разделитель строк, символ = или символ (
!!	Предыдущая команда
!*n*	Команда с номером *n* в историческом списке
!-*n*	*n*-я предыдущая команда
!*строка*	Самая последняя командная строка, начинающаяся со *строки*
!?*строка* [?]	Самая последняя команда, содержащая *строку*. Последний символ ? является необязательным
!#	Текущая команда (в том виде, в котором она набрана до этого момента)
!{*событие*}	*Событие* — это указатель события. Фигурные скобки изолируют событие от окружающего текста. Например, !{-3}3 — это третья из самых последних выполненных команд, за которой следует 3

Указатели слов

Указатель слова определяет в событии слово (лексему) или ряд слов. (Указатели слов перечислены в табл. 8.9.) Слова нумеруются с нуля (первое слово в строке обычно команда), затем идет слово 1 (первое слово, следующее за командой), и заканчивается нумерация числом *n* (последним словом в строке).

Чтобы определить конкретное слово из предыдущего события, нужно после указателя события (например, !14) поставить двоеточие и номер слова в этом событии. Например, запись !14:3 указывает на третье за командой слово из события 14. На первое слово, следующее за командой (слово номер 1), можно указать с помощью знака вставки (^), а на последнее — с помощью знака доллара ($). Можно указать диапазон слов, разделяя два указателя дефисом.

```
72 $ echo apple grape orange pear
apple grape orange pear
73 $ echo !72:2
echo grape
grape
74 $ echo !72:^
echo apple
apple
75 $ !72:0 !72:$
echo pear
pear
76 $ echo !72:2-4
echo grape orange pear
grape orange pear
77 $ !72:0-$
echo apple grape orange pear
apple grape orange pear
```

Как показано в следующем примере, символы !$ ссылаются на последнее слово предыдущего события. Это сокращение можно использовать, к примеру, для редактирования только что выведенного утилитой cat файла:

```
$ cat report.718
...
$ vim !$
vim report.718
...
```

Если событие содержит отдельную команду, то номера слов соответствуют номерам аргументов. Если событие содержит более одной команды, это соответствие для всех команд после первой не отвечает действительности. В следующем примере событие 78 содержит две команды, разделенные точкой с запятой, поэтому оболочка выполняет их последовательно; точка с запятой является словом с номером 5.

```
78 $ !72 ; echo helen zach barbara
echo apple grape orange pear ; echo helen zach barbara
apple grape orange pear
helen zach barbara
79 $ echo !78:7
echo helen
helen
80 $ echo !78:4-7
echo pear ; echo helen
pear
helen
```

Таблица 8.9. Указатели слов

Указатель	Значение
N	*n*-е слово. Обычно слово 0 представляет имя команды
^	Первое слово (после имени команды)
$	Последнее слов
m–*n*	Все слова от слова номер *m* и до слова номер *n*; если опустить *m*, то по умолчанию берется значение 0 (0-*n*
*n**	Все слова от слова номер *n* и до последнего слова
*	Все слова за исключением имени команды. То же, что и 1*
%	Слово, соответствующее самому последнему поиску ?*строка*?

Модификаторы

Иногда может потребоваться изменить какой-нибудь аспект повторно вызываемого события. Возможно, была введена сложная командная строка, в которой допущена опечатка или дано неверное имя или в ней нужно указать другой аргумент. Событие или слово события можно модифицировать, поставив один или несколько модификаторов после указателя слова или после указателя события, если отсутствует указатель слова. Перед каждым модификатором следует поставить двоеточие (:).

Модификатор подстановки

В следующем примере *модификатор подстановки* исправляет опечатку в предыдущем событии:

```
$ car /home/zach/memo.0507 /home/max/letter.0507
bash: car: command not found          Команда не найдена
$ !!:s/car/cat
cat /home/zach/memo.0507 /home/max/letter.0507
...
```

Модификатор подстановки имеет следующий синтаксис:

```
[g]s/старое/новое/
```

где *старое* — исходная строка (но не регулярное выражение), а *новое* — это строка, которая заменить *старое*. Модификатор подстановки осуществляет замену первого появления строки *старое* строкой *новое*. Если перед s поставить g (как в gs/*старое*/*новое*/), будет произведена глобальная замена всех появлений строки *старое*. Хотя в примерах используется разделитель /, можно взять любой символ, не встречающийся в строках *старое* или *новое*. Если сразу же за командой будет нажата клавиша ВВОД, заключительный разделитель можно не ставить. Как и при использовании команды Подставить в редакторе vim, механизм истории заменяет знак амперсанда (&) в строке *новое* строкой *старое*. Пустую строку *старое* (s//*новое*/) оболочка заменяет предыдущей строкой *старое* или строкой команды, которую искали с помощью команды ?*строка*?.

Быстрая подстановка

Быстрая подстановка является сокращенной формой модификатора подстановки. Она используется для повторного вызова самого последнего события, в котором нужно изменить часть его текста. Символом быстрой подстановки является знак вставки (^). Например, команда

```
$ ^старое^новое^
```

Приводит к такому же результату, что и команда

```
$ !!:s/старое/новое/
```

Таким образом подстановка в предыдущем событии cat вместо car может быть введена как

```
$ ^car^cat
cat /home/zach/memo.0507 /home/max/letter.0507
```

Если сразу же за вводом команды будет нажата клавиша ВВОД, то завершающий знак вставки можно опустить. Как и при всех остальных подстановках в командной строке, оболочка выведет командную строку в том виде, который она приобрела после подстановки.

Другие модификаторы

Модификаторы (кроме модификатора подстановки) осуществляют простое редактирование части события, выбранного указателем события и дополнительно указателями слов. Можно использовать сразу несколько модификаторов, предваряя каждый из них двоеточием (:).

В следующей серии команд для вывода списка имен файлов используется команда ls, затем команда повторяется без выполнения (модификатор p), после чего повторяется последняя команда с удалением последней части имени (модификатор h), которая также не выполняется:

```
$ ls /var/log/messages
/var/log/messages
$ !!:p
ls /var/log/messages
$ !!:h:p
ls /var/log
```

В табл. 8.10 приведены модификаторы событий, кроме модификатора подстановки.

Таблица 8.10. Модификаторы событий

Модификатор	Функция
e (extension — расширение)	Удаление всего, кроме расширения имени файла
h (head — заголовок)	Удаление последней части имени
p (print-not — не публиковать)	Вывод команды без ее выполнения
q (quote — цитировать)	Цитирование подстановки для предотвращений дальнейших подстановок на ее основе

продолжение ⇩

Таблица 8.10 (*продолжение*)

Модификатор	Функция
r (root — основа)	Удаление расширения имени файла
t (tail — окончание)	Удаление всех элементов имени, кроме последнего
X	То же действие, что и у q, но цитирует каждое слово в подстановке в отдельности

Библиотека Readline

Редактирование командной строки в оболочке Bourne Again Shell реализуется с помощью библиотеки Readline, которая доступна любому приложению, написанному на языке C. Любое приложение, использующее библиотеку Readline, поддерживает строчное редактирование, согласующееся с тем, которое предоставляется bash. Программы, использующие Readline (включая bash), считывают файл ~/.inputrc (см. раздел «Файл .inputrc: Конфигурирование библиотеки Readline» на с. 338) для получения информации о клавишных привязках и настройках конфигурации. Для выключения редактирования командной строки в bash используется ключ `--noediting`.

Режим vi При использовании в bash библиотеки Readline можно выбрать один из двух режимов редактирования: emacs или vi(m). Оба режима предоставляют множество команд, доступных в автономных версиях редакторов emacs и vim. Кроме того, для перемещений можно использовать клавиши стрелок. Перемещения вверх и вниз приводят к переходам назад и вперед по историческому списку. Кроме того, библиотека Readline предоставляет несколько видов интерактивного автозавершения слов (см. раздел «Команды автозавершения библиотеки Readline» на с. 336). По умолчанию используется режим emacs, из которого с помощью следующей команды можно переключиться в режим vi:

```
$ set -o vi
```

Режим emacs Следующая команда приводит к обратному переключению в режим emacs:

```
$ set -o emacs
```

Режим редактирования vi

Перед началом работы следует убедиться, что оболочка находится в режиме vi.

При вводе команд bash в режиме редактирования vi, вы находитесь в Режиме ввода (см. раздел «Режимы ввода команд и текста» на с. 172). Если при вводе команды до нажатия клавиши ВВОД будет замечена ошибка, можно нажать клавишу ESCAPE и перейти в Командный режим редактора vim. Его настройка отличается от настройки исходного Командного режима автономного редактора vim. Тем не менее в командном режиме для редактирования командной строки можно пользоваться многими командами vim. Это похоже на использование vim для редактирования на экране копии файла истории, имеющем место только для одной команды. При использовании команды k или СТРЕЛКИ ВВЕРХ для перемещения вверх на одну строку вы получаете предыдущую команду. Если после этого воспользоваться командой j или СТРЕЛКОЙ

ВНИЗ для перемещения вниз на одну строку, вы вернетесь к исходной команде. Для использования клавиш k и j с целью перехода между командами нужно находиться в Командном режиме; клавиши стрелок можно использовать как в командном режиме, так и в режиме ввода.

АВТОНОМНЫЙ РЕДАКТОР НАЧИНАЕТ РАБОТУ В КОМАНДНОМ РЕЖИМЕ

Автономный редактор vim начинает свою работу в Командном режиме, а vim-редактор командной строки начинает работу в Режиме ввода. Если команды выводят символы и не работают, значит, вы в Режиме ввода. Нажмите ESCAPE и введите команду снова.

Кроме команд позиционирования курсора можно использовать команду поиска в обратном направлении (?), за которой нужно ввести строку поиска, чтобы найти нужную команду, просматривая назад исторический список на наличие самой последней команды, в которой содержится искомая строка. Если вы перешли по историческому списку назад, то для поиска самой последней команды в прямом направлении можно воспользоваться прямым слэшем (/). В отличие от строк поиска в автономном редакторе vim, эти строки поиска не могут содержать регулярных выражений. Но строку поиска можно начать со знака вставки (^), чтобы заставить оболочку обнаруживать команды, которые начинаются со строки поиска. Как и в редакторе vim, нажатие клавиши n после успешного поиска приводит к поиску следующего вхождения той же строки. Для доступа к событию в историческом списке можно также воспользоваться номером события. Находясь в командном режиме (для перехода в который нужно нажать ESCAPE), введите номер события, а за ним нажмите клавишу G для перехода к событию с данным номером.

При использовании команд /, ? или G для перемещения к командной строке вы находитесь в командном режиме, а не в режиме ввода: вы можете отредактировать команду или нажать ВВОД для ее выполнения. Когда будет выведена команда, которую нужно отредактировать, можно изменить командную строку, используя команды редактирования командного режима vim, например x (удалить символ), r (заменить символ), ~ (сменить регистр) и . (повторить последнее изменение). Чтобы перейти в режим ввода, следует воспользоваться любой из команд Вставить (i, I), Добавить (a, A), Заменить (R) или Изменить (c, C). Для выполнения команды не нужно возвращаться в командный режим; следует просто нажать ВВОД, даже если курсор находится на середине командной строки. Сводка vim-команд приведена в разделе «Заключение» на с. 214.

Режим редактирования emacs

В отличие от редактора vim в emacs не используются режимы работы. При работе с ним не нужно переключаться между командным режимом и режимом ввода, поскольку большинство команд emacs является управляющими символами (см. раздел «Основные команды редактирования» на с. 231), позволяющими emacs отличать ввод от команд. Как и vim, редактор командной строки emacs предоставляет команды для перемещения курсора по командной строке и по командам исторического списка, а также команды для внесения изменений в команду или в ее часть. Но в некоторых случаях редактор командной строки emacs отличается от автономного редактора emacs.

В emacs перемещение курсора осуществляется за счет использования команд, в которых используется как клавиша CONTROL, так и клавиша ESCAPE. Чтобы пере-

местить курсор в командной строке на один символ назад, следует нажать комбинацию CONTROL+B. А для перемещения курсора на один символ вперед следует нажать комбинацию CONTROL+F. Как и в vim, перед этими перемещениями можно задать их количество. Чтобы использовать количественный показатель, сначала нужно нажать клавишу ESCAPE; в противном случае введенные числа окажутся в командной строке.

Как и vim, редактор emacs предоставляет команды для перемещения по словам и строкам. Чтобы переместить курсор в командной строке на одно слово назад или вперед, нужно нажать последовательность клавиш ESCAPE b или ESCAPE f. Чтобы переместить курсор на несколько слов, используя заданное количество, нужно нажать ESCAPE, затем ввести количество, а затем соответствующую escape-последовательность. Для перемещения курсора в начало строки нужно нажать комбинацию CONTROL+A; а чтобы переместить его в конец строки — комбинацию CONTROL+E. Чтобы переместить курсор к следующему экземпляру символа c, нужно нажать CONTROL+X CONTROL+F, а за ними клавишу c.

Можно добавить текст к командной строке, переместив курсор на позицию, с которой нужно ввести текст, и набрав нужный текст. Для удаления текста нужно поставить курсор справа от тех символов, которые нужно удалить, и нажать клавишу стирания (см. раздел «Удаление символа» на с. 48) по одному разу для каждого символа, который нужно удалить.

КОМБИНАЦИЯ CONTROL+D МОЖЕТ ЗАВЕРШИТЬ СЕАНС РАБОТЫ С ОБОЛОЧКОЙ

Если нужно удалить символ непосредственно под курсором, нажмите CONTROL+D. Если нажать CONTROL+D в начале строки, это может завершить ваш сеанс работы с оболочкой.

Если нужно удалить всю командную строку, следует набрать символ удаления строки (см. главу 2, раздел «Удаление строки»). Этот символ можно набрать, когда курсор находится в любом месте строки. Если нужно удалить текст от позиции курсора до конца строки, следует воспользоваться комбинацией CONTROL+K. Сводка команд редактора emacs приведена в разделе «Заключение» главы 7.

Команды автозавершения библиотеки Readline

Для завершения слов, вводимых в командную строку, нужно воспользоваться клавишей TAB. Это свойство, называемое автозавершением, работает в обоих режимах редактирования, как в vi, так и в emacs, и похоже на свойство автозавершения, доступное в оболочке tcsh. Доступны несколько видов автозавершения. Какой из них вы используете, зависит от того, какая часть командной строки вводится на момент нажатия клавиши TAB.

Автозавершение команды

Если вводится имя команды (обычно это первое слово командной строки), при нажатии клавиши TAB инициируется *автозавершение команды*, при котором bash ищет команду, чье имя начинается с набранной части слова. Если команд, начинающихся со введенных символов нет, bash выдает звуковой сигнал. Если существует одна команда с таким началом, bash завершает имя этой команды. Если есть более одного варианта, то в режиме vi оболочка bash не делает ничего, а в режиме emacs выдает звуковой сигнал. Повторное нажатие клавиши TAB заставляет bash вывести список команд,

чьи имена начинаются с набранного вами префикса, и дает возможность продолжить набор имени команды.

В следующем примере пользователь набрал bz и нажал клавишу TAB. Оболочка выдала звуковой сигнал (пользователь работает в режиме emacs), чтобы показать, что с символов bz начинается сразу несколько команд. Пользователь еще раз нажал клавишу TAB, чтобы заставить оболочку вывести список команд, начинающихся с bz, за которым следует набранная до этого времени командная строка:

```
$ bz ▸TAB (сигнал) ▸TAB
bzcat       bzdiff       bzip2         bzless
bzcmp       bzgrep       bzip2recover  bzmore
$ bz■
```

После этого пользователь набрал c и дважды нажал клавишу TAB. Оболочка вывела две команды, начинающиеся с символов bzc. Пользователь набрал a, поле чего нажал клавишу TAB. Теперь уже оболочка завершила команду, поскольку с символов bzca начинается только одна команда.

```
$ bzc▸TAB (сигнал) ▸TAB
bzcat  bzcmp
$ bzca ▸TAB ▸t ■
```

Автозавершение имени

Автозавершение имени, в котором также используются клавиши TAB, позволяет набрать часть имени и дать возможность bash предоставить оставшуюся часть. Если часть уже набранного имени достаточна для определения уникального имени, bash выводит это имя. Если этой части соответствует более одного имени, bash дополняет имя до того места, где нужно сделать выбор, позволяя продолжить набор.

Если вводить полное имя, включающее простое имя файла, и нажать клавишу TAB, оболочка выдаст звуковой сигнал (если она находится в режиме emacs, в режиме vi звукового сигнала не будет). Затем она проведет расширение командной строки, насколько это возможно.

```
$ cat films/dar ▸TAB (сигнал) cat films/dark_■
```

У каждого имени файла в каталоге films, который начинается с dar, следующими символами являются k_, поэтому bash не может больше расширить строку, пока не будет сделан выбор из имеющихся файлов. Оболочка оставляет курсор сразу же после символа _. Теперь можно продолжить ввод имени или дважды нажать клавишу TAB. В последнем случае bash выдает звуковой сигнал, выводит список на выбор, заново выводит командную строку и снова оставляет курсор сразу же после символа _.

```
$ cat films/dark_ ▸TAB (сигнал) ▸TAB
dark_passage  dark_victory
$ cat films/dark_■
```

Когда будет введено достаточно информации, чтобы можно было выделить один из двух возможных файлов, и нажата клавиша TAB, bash выведет уникальное имя. Если *после* _ ввести p, а затем нажать TAB, оболочка завершит командную строку:

```
$ cat films/dark_p ▸TAB ▸ assage
```

Поскольку неоднозначность устранена, оболочка добавляет пробел, позволяющий завершить набор командной строки или просто нажать ВВОД для выполнения

команды. Если завершенное полное имя является каталогом, то вместо пробела bash добавит прямой слэш (/).

Автозавершение переменной

Если при наборе имени переменной нажать клавишу TAB, это приведет к попытке bash провести автозавершение этого имени. В том случае, если однозначно определить это имя будет невозможно, двойным нажатием клавиши TAB можно будет вывести список возможных вариантов:

```
$ echo $HO ▸TAB ▸TAB
$HOME       $HOSTNAME   $HOSTTYPE
$ echo $HOM ▸TAB ▸E
```

ПРИ НАЖАТИИ КЛАВИШИ ВВОД ПРОИСХОДИТ ВЫПОЛНЕНИЕ КОМАНДЫ

Нажатие клавиши ВВОД заставит оболочку выполнить команду независимо от того, в каком месте командной строки находится курсор.

Файл .inputrc: Конфигурирование библиотеки Readline

Для получения инициализирующей информации оболочка Bourne Again Shell и другие программы, использующие библиотеку Readline, считывают файл, указанный в переменной среды окружения `INPUTRC`. Если значение этой переменной не установлено, программы считывают файл ~/.inputrc. При этом имеющиеся в файле пустые строки и строки, начинающиеся со знака решетки (#), игнорируются.

Переменные

Для управления поведением библиотеки Readline в файле .inputrc можно установить значения переменных, воспользовавшись следующим синтаксисом:

`set` *`переменная значение`*

В табл. 8.11 перечислены некоторые переменные и значения, которые могут для них использоваться. Полный перечень этих переменных можно найти на man- или info-странице, посвященной bash.

Таблица 8.11. Переменные библиотеки Readline

Переменная	Ее действие
`editing-mode`	Установка значения в `vi` запускает Readline в режиме vi. Установка в `emacs` Readline в режиме emacs (используется по умолчанию). Аналогично действию команд оболочки `set -o vi` и `set -o emacs` (см. раздел «Библиотека Readline» на с. 334)
`horizontal-scroll-mode`	Установка значения в `on` приводит к тому, что длинные строки распространяются за правой отображаемой области. Перемещение курсора вправо от крайней правой позиции отображаемой области приводит к смещению строки влево и выводу следующих символов строки. Если переместить курсор назад за крайнюю левую позицию, это приведет к обратному смещению строки. По умолчанию устанавливается значение `off`, заставляющее длинные строки отображаться сразу на нескольких строках экрана

Переменная	Ее действие
mark-directories	Установка значения в off заставляет Readline не ставить слэш (/) в конце имен каталогов при их автозавершении. По умолчанию устанавливается значение o
mark-modified-lines	Установка значения в on заставляет Readline ставить перед измененными историческими записями знак звездочки. По умолчанию устанавливается значение off

Привязки клавиш

Можно указать привязки, которые отобразят последовательности нажатий клавиш на команды Readline, позволяющие изменить или расширить привязки, используемые по умолчанию. Как и в редактор emacs, в библиотеку Readline включено множество команд, привязанных к клавишным последовательностям. Чтобы использовать команду, не имеющую привязки, ее можно отобразить, используя следующие формы:

клавиша: *имя_команды*
"*клавишная_последовательность*": *имя_команды*

В первой форме используется имя одиночной клавиши. Например, CONTROL+U будет написано как control-u. Эту форму можно использовать для привязки команд к одиночным клавишам.

Во второй форме указывается строка, описывающая последовательность клавиш, которая будет привязана к команде. Здесь для представления специальных клавиш можно использовать escape-последовательности с обратными слэшами в emacs-стиле: CONTROL (\C), META (\M) и ESCAPE (\e). Указать обратный слэш можно путем его отключения другим обратным слэшем: \\. Аналогично, обратным слэшем могут быть отключены двойные или одинарные кавычки: \" или \'.

Команда kill-whole-line, доступная только в режиме emacs, удаляет текущую строку. Чтобы привязать эту команду (которая по умолчанию остается непривязанной) к клавишной последовательности CONTROL+R, нужно поместить в файл .inputrc следующую команду:

```
control-r: kill-whole-line
```

Команда bind Чтобы вывести список всех команд Readline, нужно ввести команду bind -Ps. Если команда привязана к клавишной последовательности, то эта последовательность отображается на экране. Команды, которые можно использовать в vi-режиме, начинаются с символов vi. Например, команды vi-next-word и vi-prev-word перемещают курсор соответственно к началу следующего или предыдущего слова. Команды, не начинающиеся на vi, доступны, как правило, в emacs-режиме.

Чтобы определить, какая клавишная последовательность привязана к команде, нужно воспользоваться командой bind -q:

```
$ bind -q kill-whole-line
kill-whole-line can be invoked via "\C-r".
  kill-whole-line может быть вызвана последовательностью "\C-r".
```

Можно также привязать текст, заключив его в двойные кавычки (только в emacs-режиме):

```
"QQ": "The Linux Operating System"
```

Эта команда заставляет bash при вводе QQ вставить текст The Linux Operating System.

Условная структура

Используя директиву `$if`, можно выбрать часть файла .inputrc на основе выполнения заданных условий. Условная структура имеет следующий синтаксис:

```
$if тест[=значение]
        команды
    [$else
        команды]
$endif
```

где *тест* — это `mode`, `term` или `bash`. Если *тест* равен *значению* (или если *тест* имеет истинное значение, когда *значение* не определено) — эта структура выполняет первый набор *команд*. Если *тест* не равен *значению* (или если он имеет ложное значение, когда *значение* не определено), эта структура приводит к выполнению второго набора *команд*, если он имеется, или происходит выход из структуры, если его нет.

Работа директивы `$if` основана на трех типах доступных ей тестов.

1. Можно провести тест на определение текущего установленного режима.

   ```
   $if mode=vi
   ```

   Предыдущий тест даст истинный результат, если текущим режимом использования библиотеки Readline является vi, и ложный результат в противном случае. Можно тестировать режим на значение `vi` или значение `emacs`.

2. Можно протестировать тип терминала.

   ```
   $if term=xterm
   ```

   Предыдущий тест даст истинный результат, если значение переменной `TERM` равно `xterm`. Можно тестировать переменную `TERM` на любое значение.

3. Можно протестировать имя приложения.

   ```
   $if bash
   ```

   Предыдущий тест даст истинный результат, когда запущена программа bash, а не другая программа, использующая библиотеку Readline. Можно проводить тест на любое имя приложения.

Благодаря этим тестам можно настраивать использование библиотеки Readline на основе текущего режима, типа терминала и используемого приложения. Они предоставляют широкие возможности и гибкость использования библиотеки Readline при работе с bash и другими программами.

Следующие команды в файле .inputrc приводят к тому, что комбинация CONTROL+Y перемещает курсор к началу следующего слова, независимо от того, в каком режиме работает bash, в vi или в emacs:

```
$ cat ~/.inputrc
set editing-mode vi
$if mode=vi
        "\C-y": vi-next-word
    $else
        "\C-y": forward-word
$endif
```

Поскольку bash при запуске считывает предыдущую условную структуру, в файле .inputrc можно настроить режим редактирования. Интерактивная смена режимов с помощью команды set не приведет к изменению привязки комбинации CONTROL+Y.

Дополнительные сведения о библиотеке Readline можно получить, открыв man-страницу, посвященную bash, и задав команду /^READLINE, которая ищет слово READLINE в начале строки.

ЕСЛИ КОМАНДЫ READLINE НЕ РАБОТАЮТ, НУЖНО ВЫЙТИ ИЗ СИСТЕМЫ И СНОВА В НЕЕ ВОЙТИ

Оболочка Bourne Again Shell считывает файл ~/.inputrc при входе в систему. После внесения в этот файл изменений, нужно выйти из системы и снова в нее войти, и только тогда эти изменения вступят в силу.

Псевдонимы

Псевдоним (alias) — это имя (обычно короткое), которое оболочка переводит в другое (обычно более длинное) имя или сложную команду. Псевдонимы позволяют определять новые команды путем подстановки строки вместо первой лексемы простой команды. Обычно они размещены в файлах запуска ~/.bashrc (bash) или ~/.tcshrc (tcsh), поэтому доступны интерактивным подчиненным оболочкам.

При работе с bash для встроенной команды alias используется следующий синтаксис:

```
alias [имя[=значение]]
```

А при работе в tcsh — следующий синтаксис:

```
alias [имя[ значение]]
```

В синтаксисе bash пробелов вокруг знака равенства быть не должно. Если *значение* содержит пробелы или символы табуляции, его нужно взять в кавычки. В отличие от псевдонимов tcsh, псевдонимы bash не воспринимают в *значении* аргументов из командной строки. Если необходимо использовать аргументы, следует воспользоваться функцией bash (см. раздел «Функции» на с. 363).

Псевдоним не заменяет свое собственное содержимое, что исключает возможность бесконечной рекурсии при работе, к примеру, с такими псевдонимами:

```
$ alias ls='ls -F'
```

Псевдонимы можно вкладывать друг в друга. В неинтерактивных (то есть в сценариях оболочки) псевдонимы не работают. Чтобы просмотреть список текущих псевдонимов, нужно ввести команду alias. Чтобы посмотреть на псевдоним с конкретным именем, нужно за командой alias ввести имя псевдонима. Для удаления псевдонима можно воспользоваться встроенной командой unalias.

Когда встроенная команда alias вводится без аргументов, оболочка выводит список всех определенных псевдонимов:

```
$ alias
alias ll='ls -l'
alias l='ls -ltr'
alias ls='ls -F'
alias zap='rm -i'
```

В большинстве дистрибутивов Linux определяются по крайней мере несколько псевдонимов. Чтобы посмотреть список действующих псевдонимов, нужно ввести команду alias. Ненужные псевдонимы из файлов загрузки можно удалить.

Сравнение одинарных и двойных кавычек в псевдонимах

Когда псевдонимы включают переменные, то выбор в их синтаксисе одинарных или двойных кавычек имеет важное значение. Если *значение* взято в двойные кавычки, то любые переменные, появляющиеся в *значении*, при создании псевдонима расширяются. Если заключить *значение* в одинарные кавычки, переменные не расширяются до тех пор, пока псевдоним не будет использован. Это различие иллюстрируется в следующих примерах.

Являющаяся ключевым словом переменная PWD содержит имя рабочего каталога. При работе в своем личном каталоге пользователь Max создает два псевдонима. Поскольку при создании псевдонима dirA он использует двойные кавычки, оболочка при его создании подставляет значение рабочего каталога. Команда alias dirA выводит псевдоним dirA и показывает, какие были произведены подстановки:

```
$ echo $PWD
/home/max
$ alias dirA="echo Рабочий каталог - $PWD"
$ alias dirA
alias dirA='echo Рабочий каталог - /home/max'
```

Когда Max создает псевдоним dirB, он использует одинарные кавычки, которые удерживают оболочку от расширения переменной $PWD. Команда alias dirB показывает, что псевдоним dirB по-прежнему содержит нерасширенную переменную $PWD:

```
$ alias dirB='echo Рабочий каталог - $PWD'
$ alias dirB
alias dirB='echo Рабочий каталог - $PWD'
```

После создания псевдонимов dirA и dirB Max использует команду cd, чтобы сделать рабочим каталог cars и ввести в качестве команд каждый из псевдонимов. Псевдоним, при создании которого использовались двойные кавычки, выводит в качестве рабочего имя того каталога, в котором он создал этот псевдоним (что неверно). А псевдоним dirB выводит правильное имя рабочего каталога:

```
$ cd cars
$ dirA
Рабочий каталог - /home/max
$ dirB
Рабочий каталог - /home/max/cars
```

КАК УДЕРЖАТЬ ОБОЛОЧКУ ОТ ВЫЗОВА ПСЕВДОНИМА

Для определения, не являются ли они псевдонимами, оболочка проверяет только простые, не имеющие кавычек команды. Команды, заданные в виде относительных или абсолютных имен, оболочкой не проверяются. Если нужно ввести команду, имеющую псевдоним, использование которого нежелательно, следует предварить ее символом обратного слэша, указать абсолютное имя команды или задать команду в виде ./команда.

Примеры псевдонимов

Следующий псевдоним позволяет набирать r для повтора предыдущей команды или r abc для повторения последней командной строки, начинающейся с abc:

```
$ alias r='fc -s'
```

Если вы часто пользуетесь командой `ls -ltr`, то можно создать псевдоним, подставляющий `ls -ltr`, при вводе команды `l`:

```
$ alias l='ls -ltr'
$ l
total 41
-rw-r--r--  1 max   group   30015 Mar  1 2008 flute.ps
-rw-r-----  1 max   group    3089 Feb 11 2009 XTerm.ad
-rw-r--r--  1 max   group     641 Apr  1 2009 fixtax.icn
-rw-r--r--  1 max   group     484 Apr  9 2009 maptax.icn
drwxrwxr-x  2 max   group    1024 Aug  9 17:41 Tiger
drwxrwxr-x  2 max   group    1024 Sep 10 11:32 testdir
-rwxr-xr-x  1 max   group     485 Oct 21 08:03 floor
drwxrwxr-x  2 max   group    1024 Oct 27 20:19 Test_Emacs
```

Еще одно распространенное использование псевдонимов связано подстраховкой от ошибок. В следующем примере осуществляется подстановка интерактивной версии утилиты rm при вводе команды zap:

```
$ alias zap='rm -i'
$ zap f*
rm: remove 'fixtax.icn'? n
rm: remove 'flute.ps'? n
rm: remove 'floor'? n
```

Ключ `-i` заставляет rm запрашивать подтверждение на удаление каждого файла, помогая таким образом избежать нежелательного удаления. Можно также превратить `rm` в псевдоним команды `rm -i: alias rm='rm -i'`.

Псевдонимы в следующем примере заставляют оболочку подставлять `ls -l` при каждом вводе команды `ll` и подставлять `ls -F` при каждом вводе команды `ls`:

```
$ alias ls='ls -F'
$ alias ll='ls -l'
$ ll
total 41
drwxrwxr-x  2 max   group    1024 Oct 27 20:19 Test_Emacs/
drwxrwxr-x  2 max   group    1024 Aug 9 17:41 Tiger/
-rw-r-----  1 max   group    3089 Feb 11 2009 XTerm.ad
-rw-r--r--  1 max   group     641 Apr 1 2009 fixtax.icn
-rw-r--r--  1 max   group   30015 Mar 1 2008 flute.ps
-rwxr-xr-x  1 max   group     485 Oct 21 08:03 floor*
-rw-r--r--  1 max   group     484 Apr 9 2009 maptax.icn
drwxrwxr-x  2 max   group    1024 Sep 10 11:32 testdir/
```

Ключ `-F` заставляет команду ls выводить слэш (/) в конце имен каталогов и звездочку (*) в конце имен исполняемых файлов. В данном примере строка, заменяющая псевдоним `ll` (`ls -l`), сама содержит псевдоним (`ls`). Когда происходит замена псевдонима его значением, оболочка смотрит на первое слово заменяющей строки, нет ли в нем псевдонима. В предыдущем примере заменяющая строка содержит псевдоним `ls`, поэтому происходит вторая подстановка, формирующая окончательный вид команды `ls -F -l`. (Чтобы не попасть в рекурсию, `ls` в заменяющем тексте, хотя это и псевдоним, во второй раз уже не расширяется.)

Если задать перечень псевдонимов без поля *=значение* или *значение*, встроенная команда alias будет отвечать за вывод значения каждого заданного псевдонима.

Если какой-нибудь псевдоним не был определен, встроенная команда alias выдаст ошибку:

```
$ alias ll l ls zap wx
alias ll='ls -l'
alias l='ls -ltr'
alias ls='ls -F'
alias zap='rm -i'
bash: alias: wx: not found  не найден
```

Подстановки псевдонима можно избежать, поставив впереди команды-псевдонима обратный слэш (\):

```
$ \ls
Test_Emacs XTerm.ad  flute.ps  maptax.icn
Tiger    fixtax.icn  floor     testdir
```

Поскольку замена имени псевдонима его значением не изменяет всей остальной командной строки, выполняемая команда по-прежнему воспринимает любые аргументы:

```
$ ll f*
-rw-r--r--  1 max   group     641 Apr  1 2009 fixtax.icn
-rw-r--r--  1 max   group   30015 Mar  1 2008 flute.ps
-rwxr-xr-x  1 max   group     485 Oct 21 08:03 floor*
```

Удалить псевдоним можно с помощью встроенной команды unalias. После удаления псевдонима zap он больше не выводится встроенной командой alias, и при его последующем применении выводится сообщение об ошибке:

```
$ unalias zap
$ alias
alias ll='ls -l'
alias l='ls -ltr'
alias ls='ls -F'
$ zap maptax.icn
bash: zap: command not found  команда на найдена
```

Функции

Функции оболочки bash (у оболочки tcsh нет функций) похожи на сценарии оболочки тем, что в них сохраняются группы команд для последующего выполнения. Но, поскольку оболочка хранит функции в оперативной памяти компьютера, а не в файле, хранящемся на диске, доступ к ним она может получить быстрее, чем к сценариям. Оболочка также осуществляет предварительную обработку (синтаксический разбор) функции, поэтому функция запускается быстрее сценария. И наконец, функция оболочки выполняется той же самой оболочкой, которая ее вызвала. Если будет определено слишком много функций, затраты ресурсов на запуск подчиненной оболочки (как и при запуске сценария) могут стать неприемлемыми.

Функцию оболочки можно объявить в файле запуска ~/.bash_profile, в сценарии, который ее использует, или непосредственно в командной строке. Удалить функцию можно с помощью встроенной команды unset. Оболочка не сохраняет функции после выхода из системы.

УДАЛЕНИЕ ПЕРЕМЕННЫХ И ФУНКЦИЙ

Если у вас есть переменная оболочки и функция с таким же именем, то при использовании команды unset удаляется переменная оболочки. Если затем применить команду unset еще раз с тем же именем, она удалит функцию.

Для объявления функции используется следующий синтаксис:

```
[function] имя_функции ()
{
    команды
}
```

где слово function является необязательным, *имя_функции* — это имя, которое используется для вызова функции, а *команды* включают в себя перечень команд, выполняемых функцией при ее вызове. В качестве команд может быть все, что можно помещать в сценарий оболочки, включая вызовы других функций.

Открывающая фигурная скобка ({) может стоять на одной строке с именем функции. Псевдонимы и переменные расширяются в процессе чтения, а не в процессе выполнения функции. Чтобы прекратить выполнение функции, в ее теле можно использовать инструкцию break (см. раздел «Инструкции break и continue» на с. 438).

Функции оболочки могут использоваться как для сокращения объема кода, так и для определения специальных команд. Следующая функция запускает в фоновом режиме процесс по имени process с сохранением информации, которая обычно отображается процессом, в файле .process.out:

```
start_process() {
process > .process.out 2>&1 &
}
```

В следующем примере создается простая функция, отображающая дату, заголовок и список людей, вошедших в систему. Эта функция запускает те же команды, что и сценарий whoson, рассмотренный в разделе «chmod: Превращение файла в исполняемый» на с. 292. В этом примере функция вводится с клавиатуры. Символ «больше, чем» (>) является вторичным приглашением оболочки (PS2), поэтому его вводить не следует.

```
$ function whoson ()
> {
> date
>  echo "Users Currently Logged On"
> who
> }
$ whoson
Sun Aug 9 15:44:58 PDT 2009
Users Currently Logged On                    Пользователи, вошедшие в систему
hls      console      Aug  8 08:59  (:0)
max      pts/4        Aug  8 09:33  (0.0)
zach     pts/7        Aug  8 09:23  (bravo.example.com)
```

Функции в файлах запуска

Если нужно, чтобы функция whoson была всегда доступна и ее не приходилось вводить при каждом входе в систему, ее определение следует поместить в файл ~/.bash_profile. После этого нужно запустить файл .bash_profile, используя команду . (точка), чтобы изменения тут же вступили в силу:

```
$ cat ~/.bash_profile
export TERM=vt100
stty kill '^u'
whoson ()
{
date
echo "Users Currently Logged On"
who
}
$ . ~/.bash_profile
```

При вызове функции ей можно задать аргументы. Внутри функции эти аргументы будут доступны как позиционные параметры (см. раздел «Позиционные параметры» на с. 459). В следующем примере показана функция `arg1`, введенная с клавиатуры:

```
$ arg1 ( ) {
> echo "$1"
> }
$ arg1 first_arg
first_arg
```

Еще одним примером может послужить функция `switch()`, показанная в разделе «Настройка файлов запуска» на с. 286. Использование локальных и глобальных переменных внутри функций рассматривается в разделе «Функции» на с. 455.

Следующая функция позволяет экспортировать переменные, используя синтаксис оболочки tcsh. Встроенная команда env выводит список всех переменных среды окружения и их значения и проверяет корректную работу команды setenv:

```
$ cat .bash_profile
...
# setenv - на радость пользователям tcsh
function setenv()
{
        if [ $# -eq 2 ]
            then
                eval $1=$2
                export $1
            else
                echo "Usage: setenv NAME VALUE" 1>&2
        fi
}
$ . ~/.bash_profile
$ setenv TCL_LIBRARY /usr/local/lib/tcl
$ env | grep TCL_LIBRARY
TCL_LIBRARY=/usr/local/lib/tcl
```

Встроенная команда eval

Специальный параметр $# (см. раздел «$#: Количество аргументов командной строки» на с. 459) принимает значение, равное количеству аргументов командной строки. В данной функции, чтобы заставить bash дважды просканировать команду `$1=$2`, используется встроенная команда eval. Поскольку `$1=$2` начинается со знака доллара ($), оболочка рассматривает всю строку как единую лексему — команду. При осуществлении подстановки переменной именем команды становится `TCL_LIBRARY=/usr/local/lib/tcl`, что приводит к ошибке. При использовании eval второе сканирование разбивает строку на три лексемы, что приводит к корректной работе присваивания.

Управление bash: Свойства и настройки

В этом разделе объясняется, как управлять свойствами и настройками bash, используя ключи командной строки и встроенные команды set и shopt.

Ключи командной строки

Доступны два вида ключей командной строки: короткие и длинные. Короткие ключи состоят из дефиса, за которым следует буква; длинные ключи имеют два дефиса, за которыми следуют несколько символов. В командной строке, вызывающей bash, длинные ключи должны ставиться перед короткими. В табл. 8.12 перечислены некоторые наиболее востребованные ключи командной строки.

Таблица 8.12. Ключи командной строки

Ключ	Объяснение	Синтаксис
Help (справка)	Отображение сообщения об использовании	`--help`
No edit (не редактировать)	Предотвращение использования пользователями библиотеки Readline (см. раздел «Библиотека Readline» на с. 334) для редактирования командных строк в интерактивной оболочке	`--noediting`
No profile (не читать профиль)	Предотвращение чтения файлов запуска (см. раздел «Файлы запуска» на с. 285): /etc/profile, ~/.bash_profile, ~/.bash_login и ~/.profile	`--noprofile`
No rc (не читать rc)	Предотвращение чтения файла запуска ~/.bashrc (см. раздел «Файл .bashrc»). Если оболочка вызывается как sh, этот ключ устанавливается по умолчани	`--norc`
POSIX	Запуск bash в режиме POSIX	`--posix`
Version (версия)	Вывод информации о версии bash и выход	`--version`
Login (вход в систему)	Принуждение bash к запуску в режиме оболочки входа в систему	`-l` (буква «l» в нижнем регистре)
shopt	Запуск оболочки с ключом *opt* для команды shopt (см. далее). `-O` (буква «O» в верхнем регистре) устанавливает ключ; `+O` его снимает	`[±]O [`*opt*`]`
End of options (окончание ключей)	Сигнализирует в командной строке об окончании ключей. Все следующие лексемы рассматриваются как аргументы, даже если они начинаются с дефиса (-)	`--`

Свойства оболочки

Поведением оболочки Bourne Again Shell можно управлять за счет включения и выключения ее свойств. Встроенная команда set управляет одной группой свойств, а встроенная команда shopt управляет другой группой. Многими свойствами можно также управлять из командной строки, используемой для вызова bash.

СВОЙСТВА, КЛЮЧИ, ПЕРЕМЕННЫЕ?

Чтобы не запутаться в терминологии, в данной книге различные варианты поведения оболочки, которыми можно управлять, называются свойствами. Посвященные bash info-страницы называют их опциями — «options» и значениями переменных, управляющими избирательным поведением оболочки — «values of variables controlling optional shell behavior».

set ±o: Включение и выключение свойств оболочки

Имеющаяся в bash встроенная команда set (в tcsh также есть встроенная команда set, но она работает иначе) при использовании с ключом -o или +o включает, выключает определенные свойства bash и выводит их список. Например, следующая команда включает свойство noclobber (см. раздел «noclobber: Предотвращение перезаписи файлов» на с. 148):

```
$ set -o noclobber
```

А с помощью следующей команды это свойство можно выключить (по умолчанию оно выключено):

```
$ set +o noclobber
```

Команда set -o без ключа выводит список свойств, управляемых командой set, за которыми следует их состояние (включено — on или выключено — off). Команда set +o без ключа выводит список тех же свойств в форме, которую можно использовать как ввод для оболочки. Свойства bash перечислены в табл. 8.13.

shopt: Включение и выключение свойств оболочки

Встроенная команда shopt (shell option — вариант оболочки), которой нет в оболочке tcsh, включает, выключает определенные свойства bash, управляющие поведением оболочки, и выводит их список. Например, следующая команда заставляет bash включить имена файлов, которые начинаются с точки (.) при расширении нестрогих ссылок на файлы (ключ -s означает set — включить):

```
$ shopt -s dotglob
```

А с помощью следующей команды это свойство можно выключить (по умолчанию оно выключено), ключ -u означает unset — выключить:

```
$ shopt -u dotglob
```

Если в качестве единственного аргумента команде shopt дать имя свойства, то она покажет состояние его установки:

```
$ shopt dotglob
dotglob         off
```

Команда shopt без ключей и аргументов выводит список управляемых этой командой свойств с указанием их состояния. Команда shopt -s без аргумента выводит список свойств, управляемых этой командой, которые установлены или находятся во включенном состоянии — on. Команда shopt -u выводит список неустановленных или отключенных (off) свойств. Свойства bash приведены в табл. 8.13.

УСТАНОВКА СВОЙСТВ SET ±O С ИСПОЛЬЗОВАНИЕМ КОМАНДЫ SHOPT

Команду shopt можно использовать для установки и отключения свойств, которые кроме этого управляются командой set ±o. Для этого используется обычный синтаксис команды shopt с ключами –s или –u и включающий ключ –o. Следующая команда, например, включает свойство noclobber:

```
$ shopt -o -s noclobber
```

Таблица 8.13. Свойства bash

Свойство	Описание	Синтаксис	Альтернативный синтаксис
allexport	Автоматическое экспортирование всех созданных вами переменных и функций или их изменение после ввода этой команды	`set -o allexport`	`set -a`
braceexpand	Принуждение bash к расширению данных в фигурных скобках (включено по умолчанию; см. раздел «Расширение данных, заключенных в фигурные скобки» на с. 353)	`set -o braceexpand`	`set -B`
cdspell	Исправление незначительных опечаток в именах каталогов, используемых в качестве аргументов для команды cd	`shopt -s cdspell`	
cmdhist	Сохранение всех строк многострочной команды в одной исторической записи с добавлением при необходимости точек с запятой	`shopt -s cmdhist`	
dotglob	Принуждение к тому, чтобы все специальные символы оболочки в нестрогих ссылках на файлы (раздел «Генерация имени файла и расширение имени» на с. 157) соответствовали лидирующей точке в имени файла. По умолчанию специальные символы не соответствуют лидирующей точке. Имена файлов . и .. должны всегда указываться в явном виде, поскольку им не соответствует ни один шаблон	`shopt -s dotglob`	
emacs	Указание на применение режима emacs при редактировании командной строки (устанавливается по умолчанию; см. раздел «Режим редактирования emacs» на с. 335)	`set -o emacs`	
errexit	Принуждение bash к выходу при неудачном выполнении простой команды (за исключением управляющей структуры	`set -o errexit`	`set -e`
execfail	Принуждение сценария оболочки к продолжению выполнения при невозможности найти файл, заданный в качестве аргумента для команды exec. По умолчанию, когда exec не может найти файл, заданный аргументом, выполнение сценария прекращается	`shopt -s execfail`	
expand_aliases	Заставляет псевдонимы (раздел «Псевдонимы» на с. 363) расширяться (по умолчанию это свойство включено для интерактивных оболочек и выключено для неинтерактивных)	`shopt -s expand_alias`	
hashall	Заставляет bash запомнить, где находится команда, найденная с помощью переменной PATH (раздел «Переменная PATH: Указание тех мест, где оболочка ищет программы» на с. 312), включено по умолчанию	`set -o hashall`	`set -h`

продолжение ⇩

Таблица 8.13 (*продолжение*)

Свойство	Описание	Синтаксис	Альтернативный синтаксис
histappend	Заставляет bash при выходе из оболочки добавлять исторический список к файлу HISTFILE (раздел «Переменные, управляющие историей» на с. 324). По умолчанию bash переписывает этот фай	`shopt -s histappend`	
histexpand	Включение механизма расширения истории (который по умолчанию использует восклицательные знаки; см. раздел «Использование восклицательного знака (!) для ссылки на событие» на с. 329).Для отключения расширения истории это свойство нужно выключить	`set -o histexpand`	`set -H`
history	Включение истории команд (по умолчанию это свойство включено; см. раздел «История» на с. 324)	`set -o history`	
huponexit	Указание bash на отправку сигнала SIGHUP всем заданиям при выходе из интерактивной оболочки, через которую осуществлялся вход в систему	`shopt -s huponexit`	
ignoreeof	Указание на то, что bash перед выходом из нее должна получить десять символов конца файла — EOF. Используется на зашумленных коммутируемых линиях	`set -o ignoreeof`	
monitor	Включение управления заданиями (включено по умолчанию; см. раздел «Управление заданиями» на с. 299)	`set -o monitor`	`set -m`
nocaseglob	Принуждение к тому, чтобы нестрогие ссылки на файлы (раздел «Генерация имени файла и расширение имени» на с. 157) соответствовали именам файлов без учета регистра символов (по умолчанию выключено)	`shopt -s nocaseglob`	
noclobber	Помощь в предотвращении переписывания файлов (по умолчанию выключено; см. раздел «noclobber: Предотвращение перезаписи файлов» на с. 148)	`set -o noclobber`	`set -C`
noglob	Отключение расширения имен файлов (по умолчанию выключено; см. раздел «Генерация имени файла и расширение имени» на с. 157)	`set -o noglob`	`set -f`
notify	При включенном управлении заданиями (раздел «Управление заданиями» на с. 362) немедленное сообщение о статусе прекращения фоновых заданий. По умолчанию статус выводится перед следующим приглашением	`set -o notify`	`set -b`
nounset	Вывод сообщения об ошибке и выход из сценария оболочки, когда в интерактивной оболочке используется переменная unset. По умолчанию для переменной unset выводится пустое значение	`set -o nounset`	`set -u`

Свойство	Описание	Синтаксис	Альтернативный синтаксис
nullglob	Принуждение bash к расширению нестрогой ссылки на файлы (раздел «Генерация имени файла и расширение имени» на с. 157), которая не соответствует имени файла, в пустую строку. По умолчанию bash передает эти ссылки на файлы без их расширени	`shopt -s nullglob`	
posix	Запуск bash в режиме POSIX	`set -o posix`	
verbose	Вывод командных строк в том виде, в котором их считывает bash	`set -o verbose`	`set -v`
vi	Указание на применение режима vi при редактировании командной строки (раздел «Режим редактирования vi» на с. 334	`set -o vi`	
xpg_echo	Принуждение встроенной команды echo к расширению последовательностей, начинающихся с символа обратного слэша без необходимости применения ключа `-e` (раздел «echo -e» на с. 442)	`shopt -s xpg_echo`	
xtrace	Включение отладки оболочки (раздел «Отладка сценариев оболочки» на с. 428)	`set -o xtrace`	`set -x`

Обработка командной строки

При работе в интерактивном режиме или при запуске сценария оболочки bash требуется прочитать командную строку, прежде чем она сможет приступить к ее обработке — перед обработкой команды bash всегда читает как минимум одну строку. Некоторые встроенные команды bash, например if и case, а также функции и взятые в кавычки строки, распространяются на несколько строк. Когда bash распознает команду, занимающую более одной строки, то перед обработкой команды она читает ее целиком. В сеансах интерактивной работы по мере набора каждой строки многострочной команды bash выводит вторичные приглашения (PS2, по умолчанию >; см. раздел «PS2: Приглашение пользователю (вторичное)» на с. 315, пока не распознает конец команды:

```
$ echo 'hi
> end'
hi
end
$ function hello () {
> echo hello there
> }
$
```

После считывания командной строки bash применяет к ней расширение истории и замену псевдонима.

Расширение истории

В разделе «Повторное выполнение и редактирование команд» на с. 326 рассматриваются команды, которые могут вводиться для изменения повторного выполнения командных строк из исторического списка. Например, когда вводится команда !!, историческое расширение изменяет эту командную строку, и она становится такой же, как предыдущая. По умолчанию для интерактивных оболочек историческое расширение находится во включенном состоянии, оно выключается командой `set +o histexpand`. К неинтерактивным оболочкам (сценариям оболочки) историческое расширение не применяется.

Замена псевдонима

Механизм псевдонимов (см. соответствующий раздел) осуществляет замену строкой первого слова простой команды. По умолчанию использование псевдонимов включено для интерактивных оболочек и выключено для неинтерактивных. Чтобы выключить псевдонимы, нужно ввести команду `shopt -u expand_aliases`.

Синтаксический разбор и сканирование командной строки

После обработки исторических команд и псевдонимов bash не переходит к немедленному выполнению команды. В первую очередь оболочка проводит *синтаксический разбор* (изолирует символьные строки), превращая командную строку в лексемы и слова. Затем оболочка сканирует каждую лексему на наличие специальных символов и шаблонов, которые предписывают оболочке совершение определенных действий. Эти действия могут заключаться в подстановке одного слова или слов другими словами. При разборе оболочкой следующей командной строки она разбивает ее на три лексемы (`cp`, `~/letter` и `.`):

```
$ cp ~/letter .
```

После разбиения на лексемы и перед выполнение команды оболочка сканирует лексемы и выполняет расширение командной строки.

Расширение командной строки

Перед передачей командной строки вызывающей программе оба типа оболочек, интерактивные и неинтерактивные, производят преобразование, используя *расширение командной строки*. Можно использовать оболочку, не имея полного представления о расширении командной строки, но, разбираясь в данном вопросе, можно воспользоваться теми преимуществами, которые предлагаются оболочкой. В этом разделе рассматривается расширение командной строки, осуществляемое оболочкой Bourne Again Shell, а расширение командной строки, осуществляемое оболочкой TC Shell, рассматривается в главе 9, в разделе «Расширение командной строки (подстановка)» на с. 371.

Оболочка Bourne Again Shell сканирует каждую лексему в поисках расширения и подстановок различного вида в следующем порядке. Большинство этих процессов расширяют слово в отдельное слово. Только расширение данных, заключенных в фигурные скобки, разбиение слова и расширение имени могут изменить количество

слов в команде (исключение делается для расширения переменной "$@" — см. раздел «Сравнение „$*“ и „$@“» на с. 463).

1. Расширение данных, заключенных в фигурные скобки.
2. Расширение тильды.
3. Расширение параметров и переменных.
4. Арифметическое расширение.
5. Командная подстановка.
6. Разбиение слов.
7. Расширение имени.
8. Подстановка процесса.

Удаление символов цитирования

Когда bash закончит работу согласно предыдущему списку, она удаляет из командной строки одинарные кавычки, двойные кавычки и обратные слэши, которые не являются результатами расширения. Этот процесс называется удалением символов цитирования.

Порядок расширения

Порядок, в котором bash выполняет все эти шаги, влияет на интерпретацию команд. Например, если вы установили для переменной значение, похожее на инструкцию перенаправление вывода, а затем ввели команду, которая использует значение переменной для осуществления перенаправления, то от bash можно ожидать перенаправления вывода.

```
$ SENDIT="> /tmp/saveit"
$ echo xxx $SENDIT
xxx > /tmp/saveit
$ cat /tmp/saveit
cat: /tmp/saveit: No such file or directory Такой каталог или файл отсутствует
```

На самом деле оболочка не перенаправляет вывод — она распознает перенаправление ввода и вывода перед вычислением переменных. При выполнении командной строки оболочка проверяет наличие перенаправления и, не найдя такового, вычисляет переменную SENDIT. После замены имени переменной строкой > /tmp/saveit bash передает аргументы команде echo, которая исправно копирует свои аргументы в стандартный вывод. Никакого файла /tmp/saveit не создается.

В следующем разделе дается более подробное описание этапов обработки командной строки. Следует помнить, что двойные и одинарные кавычки заставляют оболочку при осуществлении расширений вести себя по-другому. Двойные кавычки разрешают расширение параметров и переменных, но подавляют другие виды расширений. Одинарные кавычки подавляют все виды расширений.

Расширение данных, заключенных в фигурные скобки

Расширение данных, заключенных в фигурные скобки, которое впервые появилось в оболочке C Shell, предоставляет удобный способ для указания имен файлов, когда не применяется расширение имени. Хотя это расширение почти всегда применяется для указания имен файлов, его механизм может быть использован для генерации

произвольных строк; оболочка не пытается соотносить запись в фигурных скобках с именами существующих файлов.

По умолчанию это расширение включено для интерактивных и неинтерактивных оболочек, выключить его можно с помощью команды `set +o braceexpand`. Оболочка также использует фигурные скобки для изоляции имен переменных (см. раздел «Фигурные скобки» на с. 308).

В следующем примере проиллюстрировано, как работает расширение данных, заключенных в фигурные скобки. Команда ls не выводит никаких выходных данных, поскольку в рабочем каталоге нет файлов. Встроенная команда echo выводит строку, сгенерированную оболочкой благодаря расширению данных, заключенных в фигурные скобки. В данном случае строка не сравнивается с именами файлов (поскольку в рабочем каталоге файлов нет).

```
$ ls
$ echo chap_{one,two,three}.txt
chap_one.txt chap_two.txt chap_three.txt
```

Оболочка расширяет в команде echo строку с запятыми в качестве разделителей, заключенную в фигурные скобки в разделенный пробелами список строк. Каждая строка из этого списка предваряется строкой `chap_`, называемой *преамбулой*, и дополняется строкой `.txt`, называемой *постскриптумом*. И преамбула, и постскриптум являются необязательными. При расширении сохраняется имеющийся в строке порядок слева направо. Для оболочки этот порядок нужен, чтобы особым образом обработать левую и правую фигурные скобки, а для того чтобы произошло расширение, внутри фигурных скобок нужна как минимум одна запятая и не должно быть никаких не взятых в кавычки пробельных символов. Расширение данных, заключенных в фигурные скобки, может быть вложенным.

Это расширение удобно использовать при длинных преамбулах или постскриптумах. В следующем примере в рабочий каталог копируются четыре файла — main.c, f1.c, f2.c и tmp.c, которые находятся в каталоге /usr/local/src/C:

```
$ cp /usr/local/src/C/{main,f1,f2,tmp}.c .
```

Это расширение можно также использовать для создания каталогов с родственными именами:

```
$ ls -F
file1  file2  file3
$ mkdir vrs{A,B,C,D,E}
$ ls -F
file1  file2  file3  vrsA/  vrsB/  vrsC/  vrsD/  vrsE/
```

Ключ `-F` заставляет команду ls вывести слэш (/) после каталога и звездочку (*) после исполняемого файла.

При попытке использования для указания каталогов вместо фигурных скобок нестрогой ссылки на файлы результат будет другим (и не тем, которого хотелось бы достичь):

```
$ rmdir vrs*
$ mkdir vrs[A-E]
$ ls -F
file1  file2  file3  vrs[A-E]/
```

Нестрогая ссылка на файлы соответствует именам существующих файлов. В предыдущем примере из-за того, что она не нашла имен файлов, соответствующих vrs[A-E], bash передает нестрогую ссылку на файлы команде mkdir, которая создает каталог с именем этой ссылки. Квадратные скобки в нестрогих ссылках на файлы рассмотрены в разделе «Специальные символы [ ]» на с. 160.

Расширение тильды

В главе 4 была представлена краткая запись для указания вашего личного каталога или личного каталога другого пользователя. В этом разделе дается более подробное описание *расширения тильды*.

Тильда (~) является специальным символом, который появляется в начале лексемы в командной строке. Когда bash видит тильду в этой позиции, она рассматривает следующую строку символов вплоть до первого слэша (/) или до конца слова, если слэш отсутствует, как возможное имя пользователя. Если это возможное имя пользователя является нулевой строкой (то есть если тильда появляется как самостоятельное слово или если сразу за ней следует слэш), оболочка подставляет вместо тильды значение переменной HOME. В следующем примере показывается это расширение, где последняя команда копирует файл по имени letter из личного каталога пользователя Max в рабочий каталог:

```
$ echo $HOME
/home/max
$ echo ~
/home/max
$ echo ~/letter
/home/max/letter
$ cp ~/letter .
```

Если идущая за тильдой строка символов формирует допустимое имя пользователя, оболочка подставляет вместо тильды и имени пользователя путь к личному каталогу, связанному с этим именем пользователя. Если строка не пустая и не является допустимым именем пользователя, оболочка не делает никаких подстановок:

```
$ echo ~zach
/home/zach
$ echo ~root
/root
$ echo ~xx
~xx
```

Тильды также используются при работе со стеком каталогов (см. раздел «Работа со стеком каталогов» на с. 302). Кроме того, комбинация символов ~+ является синонимом переменной PWD (содержащей имя рабочего каталога), а комбинация символов ~- является синонимом переменной OLDPWD (содержащей имя того каталога, который только что был рабочим).

Расширение параметров и переменных

Расширение параметра или переменной представлено в командной строке знаком доллара ($), за которым не следует открывающая круглая скобка. *Параметры* включают в себя как параметры командной строки, или позиционные параметры (см. раз-

дел «Позиционные параметры» на с. 459), так и специальные параметры (см. раздел «Специальные параметры» на с. 457). *Переменные* включают в себя как переменные, созданные пользователем (см. раздел «Пользовательские переменные» на с. 306), так и переменные, являющиеся ключевыми словами (см. раздел «Переменные, являющиеся ключевыми словами» на с. 311).

На посвященных bash man- и info-страницах этого различия не делается. Параметры и переменные не подвергаются расширению, если они заключены в одинарные кавычки или если лидирующий знак доллара отключен (то есть впереди него поставлен обратный слэш). Если параметры и переменные заключены в двойные кавычки, то оболочка осуществляет их расширение.

Арифметическое расширение

Оболочка осуществляет *арифметическое расширение*, вычисляя арифметическое выражение и заменяя его результатом вычисления. Сведения об арифметическом расширении в оболочке tcsh даны в разделе «Числовые переменные» на с. 386. При работе с bash для арифметических расширений используется следующий синтаксис:

$((выражение))

Оболочка вычисляет выражение и заменяет *$((выражение))* результатом вычисления. Этот синтаксис похож на синтаксис команды подстановки [*$(...)*] и выполняет параллельную функцию. Элемент *$((выражение))* можно использовать в командной строке как аргумент команды или вместо любого числового значения.

Правила составления выражения такие же, как в языке программирования C; доступны все стандартные арифметические операторы языка C (см. табл. 10.8). Арифметические вычисления в bash ведутся с использованием целых чисел. Если же целочисленные переменные (см. раздел «Целые числа» на с. 311) или настоящие целые числа не используются, то для производства арифметических вычислений оболочка должна превращать переменные, имеющие строковые значения, в целые числа.

Внутри выражения ставить знак доллара ($) перед именами переменных не нужно. В следующем примере команда read (раздел «read: Получение пользовательского ввода» на с. 466) присваивает ответ пользователя на вопрос о возрасте переменной age, а арифметическое выражение определяет, сколько лет осталось до шестидесятилетнего возраста:

```
$ cat age_check
#!/bin/bash
echo -n "How old are you? "
read age
echo "Wow, in $((60-age)) years, you'll be 60!"

$ ./age_check
How old are you? 55              Сколько вам лет? 55
Wow, in 5 years, you'll be 60! Надо же, через 5 лет вам будет 60!
```

Выражение не нужно брать в кавычки, поскольку bash не осуществляет в нем расширения имени файла. Как показано в следующем примере, эта особенность упрощает использование звездочки (*) для операции умножения:

```
$ echo There are $((60*60*24*365)) seconds in a non-leap year.
There are 31536000 seconds in a non-leap year.  В невисокосном году 31 536 000 секунд
```

В следующем примере для подсчета количества страниц, необходимого для распечатки содержимого файла letter.txt, используются команды wc, cut, арифметическое расширение и командная подстановка (см. раздел «Командная подстановка» на с. 358). Вывод утилиты wc (word count — количество слов) (см. часть V), используемой с ключом –l, является количеством строк в файле, в столбцах (символьных позициях) от 1 и до 4, за которыми идут пробел и имя файла (этот вывод следует за первой командой). Утилита cut (см. часть V) с ключами `-c1-4` извлекает первые четыре столбца.

```
$ wc -l letter.txt
351 letter.txt
$ wc -l letter.txt | cut -c1-4
351
```

Знак доллара и одинарная круглая скобка заставляют оболочку осуществить командную подстановку; знак доллара и двойная круглая скобка служат индикатором арифметического расширения:

```
$ echo $(( $(wc -l letter.txt | cut -c1-4)/66 + 1))
6
```

В предыдущем примере стандартный вывод из команды wc попадает на стандартный ввод команды cut через канал. Благодаря командной подстановке вывод обеих команд заменяет в командной строке команды между `$(` и соответствующей кавычкой `)`. Затем арифметическое расширение делит полученное от них число на 66, количество строк на странице. Прибавление единицы обусловлено тем, что при целочисленном делении остатки отбрасываются.

УМЕНЬШЕНИЕ КОЛИЧЕСТВА ЗНАКОВ ДОЛЛАРА ($)

При использовании переменных внутри $((и)) знак доллара, предшествующий отдельной переменной, ставить необязательно:

```
$ x=23 y=37
$ echo $((2*$x + 3*$y))
157
$ echo $((2*x + 3*y))
```

Другой способ получения такого же результата без использования утилиты cat заключается в перенаправлении ввода в утилиту wc, вместо того чтобы заставлять wc получать ввод из файла, чье имя указано в командной строке. При перенаправлении его ввода wc не отображает имя файла:

```
$ wc -l < letter.txt
   351
```

Присваивание результата арифметического расширения переменной является общепринятой практикой:

```
$ numpages=$(( $(wc -l < letter.txt)/66 + 1))
```

Встроенная команда let

Встроенная команда let (в tcsh такая команда отсутствует) вычисляет арифметические выражения точно так же, как и синтаксическая конструкция `$(( )`. Следующая команда является эквивалентом предыдущей:

```
$ let "numpages=$(wc -l < letter.txt)/66 + 1"
```

Двойные кавычки удерживают пробелы (как те, которые можно увидеть в примере, так и те, которые получаются в результате подстановки) от разделения выражения на отдельные аргументы для команды let. Значение последнего выражения определяет выходной статус команды let. Если значение последнего выражения равно 0, выходной статус let равен 1; в противном случае он равен 0. Можно предоставить команде let несколько аргументов в одной и той же командной строке:

```
$ let a=5+3 b=7+2
$ echo $a $b
8 9
```

Когда в процессе арифметического расширения с помощью команды let и конструкции $(( ))делается ссылка на переменную, ставить впереди переменной знак доллара ($) не нужно. Тем не менее его установка считается правилом хорошего тона, позволяющим придерживаться единства представления, поскольку в большинстве мест переменная должна предваряться знаком доллара.

Командная подстановка

Командная подстановка заменяет команду выводом этой команды. В bash предпочтение для командной подстановки нужно отдавать следующему синтаксису:

```
$(команда)
```

При работе в bash можно также использовать следующий, старый синтаксис, являющийся единственным синтаксисом, разрешенным в tcsh:

```
'команда'
```

Оболочка выполняет команду в подчиненной оболочке и заменяет команду вместе с окружающими ее знаками пунктуации стандартным выводом команды.

В следующем примере оболочка выполняет команду pwd и подставляет вывод команды вместо самой команды и окружающей ее пунктуации. Затем оболочка передает вывод команды, который теперь является аргументом, команде echo, которая его выводит на экран.

```
$ echo $(pwd)
/home/max
```

В следующем сценарии выход встроенной команды pwd присваивается переменной where, а выводимое на экран сообщение содержит значение этой переменной:

```
$ cat where
where=$(pwd)
echo "You are using the $where directory."
$ ./where
You are using the /home/zach directory. Вы используете каталог /home/zach
```

Хотя в примере иллюстрируется порядок присваивания вывода команды переменной, он не имеет практической ценности. Вывод команды pwd можно осуществить и без посредничества переменной:

```
$ cat where2
echo "You are using the $(pwd) directory."
$ ./where2
You are using the /home/zach directory.
```

В следующей команде используется утилита find для поиска файлов с именем README в древе каталогов, берущем начало от рабочего каталога. Этот список файлов является стандартным выводом утилиты find и становится списком аргументов для утилиты ls.

```
$ ls -l $(find . -name README -print)
```

В следующей командной строке показан старый синтаксис *'команда'*:

```
$ ls -l 'find . -name README -print'
```

Одно из преимуществ нового синтаксиса состоит в отмене прежних запутанных правил для обработки лексем, кавычек и обратных апострофов, использовавшихся в старом синтаксисе.

Другим преимуществом нового синтаксиса в отличие от старого является возможность вложения командной подстановки. Например, с помощью следующей команды можно создать длинную распечатку всех файлов README, чей размер превышает размер файла ./README:

```
$ ls -l $(find . -name README -size +$(echo $(cat ./README | wc -c)c ) -print )
```

Попробуйте ввести эту команду после ввода команды `set -x` (см. раздел «Отладка сценариев оболочки» на с. 428), чтобы посмотреть, как bash ее расширит. Если файлов README не окажется, будет получен только лишь вывод команды `ls -l`.

Сценарии, использующие командную подстановку, показаны также в разделах «Сценарий lnks» на с. 425, «echo –e» на с. 422 и «Встроенная команда: (нулевая строка)» на с. 474.

СРАВНЕНИЕ $((И $(

Символы $((составляют отдельную лексему. Они представляют арифметическое выражение, а не командную подстановку. Поэтому, если внутри конструкции $( ) нужно использовать вызов подчиненной оболочки для группы команд, заключенной в круглые кавычки (см. раздел «( ) — средство группировка команд» на с. 298), между символами $(и следующим за ними символом (следует поставить пробел.

Разбиение слов

Результаты расширения параметров и переменных, командной подстановки и арифметического расширения являются кандидатами на разбиение слов. Используя каждый символ переменной IFS (см. раздел «Переменная IFS: Разделение полей ввода (разбиение слов)» на с. 316) в качестве возможного разделителя, bash разбивает эти кандидаты на слова или лексемы. Если значение переменной IFS не установлено, bash использует ее значения по умолчанию (ПРОБЕЛ-TAB-NEWLINE). Если значением IFS является пустая строка, bash слова не разбивает.

Расширение имени

Расширение имени (см. раздел «Генерация имени файла и расширение имени» на с. 157), также называемое генерацией имени или подстановкой, является процессом интерпретации нестрогих ссылок на файлы и подстановки соответствующего списка файловых имен. Если установлено свойство noglob (см. табл. 8.13), оболочка выполняет эту функцию при встрече нестрогой ссылки на файлы — лексемы, содержащей любой из непроцитированных символов *, ?, [или]. Если bash не может найти место-

нахождение каких-нибудь файлов, соответствующих указанному шаблону, лексема с нестрогой ссылкой на файлы остается нетронутой. Оболочка не удаляет лексему и не заменяет ее нулевой строкой, а вместо этого передает ее программе в исходном виде (исключения даны в описании свойства nullglob в табл. 8.13). А оболочка TC Shell генерирует сообщение об ошибке.

В следующем примере в первой команде echo оболочка расширяет нестрогую ссылку на файлы tmp* и передает три лексемы (tmp1, tmp2 и tmp3) команде echo. Встроенная команда echo выводит три имени файла, переданные ей оболочкой. После того как команда rm удаляет три файла tmp*, оболочка при попытке расширения не находит имен файлов, соответствующих tmp*. После этого она передает нерасширенную строку встроенной команде echo, которая выводит переданную ей строку.

```
$ ls
tmp1 tmp2 tmp3
$ echo tmp*
tmp1 tmp2 tmp3
$ rm tmp*
$ echo tmp*
tmp*
```

По умолчанию такая же команда заставляет оболочку TC Shell вывести сообщение об ошибке:

```
tcsh $ echo tmp*
echo: No match      Соответствий не найдено
```

Если не установлено свойство dotglob (см. табл. 8.13), то точка, с которой начинается имя, или точка, за которой следует слэш (/), должны соответствовать шаблону явным образом. Свойство nocaseglob (см. табл. 8.13) заставляет нестрогие ссылки на файлы соответствовать именам файлов без учета регистра символов.

Кавычки Заключение аргументов в двойные кавычки заставляет оболочку подавлять расширение имени и другие виды расширений, за исключением расширений параметров и переменных. Заключение аргументов в одинарные кавычки приводит к подавлению всех видов расширений. В следующем примере вторая команда echo показывает переменную $max, взятую в двойные кавычки, разрешающие расширение переменной. В результате оболочка расширяет переменную, заменяя ее имя значением sonar. Но в третьей команде echo, в которой используются одинарные кавычки, расширения не происходит. Поскольку ни двойные, ни одинарные кавычки не допускают расширения имени, последние две команды выводят нерасширенный аргумент tmp*.

```
$ echo tmp* $max
tmp1 tmp2 tmp3 sonar
$ echo "tmp* $max"
tmp* sonar
$ echo 'tmp* $max'
tmp* $max
```

Оболочка различает значение переменной и ссылку на переменную и не расширяет нестрогие ссылки на файлы, если они встречаются в значении переменной. Следовательно, переменной можно присвоить значение, включающее специальные символы, например звездочку (*).

Уровни расширения

В следующем примере в рабочем каталоге имеются три файла, чьи имена начинаются со слова letter. Когда переменной var присваивается значение letter*, оболочка не расширяет нестрогую ссылку на файлы, поскольку она встречается в значении переменной (в инструкции присваивания значения переменной). Никакие кавычки строку letter* не окружают, расширению препятствует само содержимое. После присваивания встроенная команда set (с помощью утилиты grep) показывает, что значение переменной var равно letter*.

```
$ ls letter*
letter1  letter2  letter3
$ var=letter*
$ set | grep var
var='letter*'
$ echo '$var'
$var
$ echo "$var"
letter*
$ echo $var
letter1 letter2 letter3
```

Три команды echo демонстрируют три уровня расширения. Когда строка $var заключена в одинарные кавычки, оболочка не производит никаких расширений и передает символьную строку $var команде echo, которая и выводит эту строку. При использовании двойных кавычек оболочка осуществляет только расширение переменной и подставляет вместо имени переменной var, перед которым стоит знак доллара, ее значение. В этой команде не производится никакого расширения имени, поскольку двойные кавычки его подавляют. В заключительной команде таких ограничений уже нет, и оболочка осуществляет подстановку значения переменной вместо ее имени, а затем (до передачи аргументов команде echo) осуществляет расширение имени.

Подстановка процесса

Особенностью оболочки Bourne Again Shell является возможность замены аргументов файловых имен процессами. Аргумент, имеющий синтаксис *<(команда)*, заставляет выполнить *команду* и записать вывод в поименованный конвейер (FIFO — первым пришел, первым вышел). Оболочка заменяет этот аргумент именем конвейера. Если затем во время обработки этот аргумент используется как имя входного файла, вывод команды считывается. По аналогии с этим, аргумент с синтаксисом *>(команда)* заменяется именем конвейера, из которого *команда* читает данные как из стандартного ввода.

В следующем примере используется утилита sort (см. раздел «sort: Отображение содержимого файла в определенном порядке» на с. 74, а также часть V) с ключом -m (merge — объединить, который правильно работает, только если входные файлы уже отсортированы) для объединения двух списков слов в один. Каждый список слов генерируется конвейером, который извлекает слова, соответствующие шаблону из файла, и сортирует слова, имеющиеся в этом списке.

```
$ sort -m -f <(grep "[^A-Z]..$" memo1 | sort) <(grep ".*aba.*" memo2 |sort)
```

Заключение

Оболочка представляет собой как командный интерпретатор, так и язык программирования. Как командный интерпретатор она выполняет команды, введенные в ответ на ее приглашение. Как язык программирования оболочка выполняет команды из файлов, называемых сценариями оболочки. При запуске оболочки она, как правило, запускает один или несколько файлов запуска.

Запуск сценария оболочки

Если предположить, что файл со сценарием оболочки находится в рабочем каталоге, можно выделить три основных способа выполнения сценария оболочки из командной строки.

1. Ввод простого имени того файла, который содержит сценарий.
2. Ввод относительного имени, включая простое имя файла, предваряемое символами ./.
3. Ввод bash или tcsh, за которыми следует имя файла.

Первая технология требует, чтобы рабочий каталог был указан в переменной PATH. Технологии 1 и 2 требуют, чтобы у вас были права на выполнение и чтение файла, содержащего сценарий. Технология 3 требует, чтобы у вас были права на чтение файла, содержащего сценарий.

Управление заданиями

Задание представляет собой одну или несколько команд, объединенных каналами. С помощью встроенной команды fg задание, запущенное в фоновом режиме, можно перевести в активный режим. Используя встроенную команду bg, можно активное задание перевести в фоновый режим, если сначала обеспечить приостановку задания путем нажатия клавиши приостановки (как правило, это CONTROL+Z). Для просмотра работающих или приостановленных заданий следует воспользоваться встроенной командой jobs.

Переменные

Оболочка позволяет определять переменные. Переменную можно объявить и инициализировать путем присваивания ей значения; объявление переменной можно удалить с помощью команды unset. По отношению к процессу, переменные обладают локальной областью видимости, если только они не экспортированы с помощью встроенной команды export (bash) или встроенной команды setenv (tcsh) с целью их доступности в дочерних процессах. Объявленные вами переменные называются пользовательскими. Оболочка определяет переменные, являющиеся ключевыми словами. Внутри сценария оболочки можно работать с параметрами командной строки (позиционными параметрами), с которыми был вызван сценарий.

Процессы

У каждого процесса есть уникальный идентификационный номер (PID), и каждый процесс выполняет отдельную Linux-команду. При вводе команды оболочка для ее выполнения ответвляет новый (дочерний) процесс, если только команда не является встроенной в саму оболочку. При работе дочернего процесса оболочка находится в так называемом приостановленном (спящем) состоянии. Если завершить командную строку символом амперсанда (&), можно запустить дочерний процесс в фоновом режиме и обойтись без спящего состояния, получив сразу же после нажатия клавиши ВВОД приглашение оболочки. Каждая команда в сценарии оболочки ответвляет отдельный процесс, каждый из которых может, в свою очередь, ответвить другие процессы. Когда процесс прекращает свою работу, он воз-

вращает свой статус выхода своему родительскому процессу. Нулевой выходной статус свидетельствует об успешном завершении, а ненулевой означает неудачное завершение работы.

История Механизм истории — это свойство, позаимствованное из оболочки C Shell, которое поддерживает список самых последних введенных командных строк, также называемых событиями, предоставляющий способ быстрого повторного выполнения предыдущих команд. Существует несколько способов работы с историческим списком, один из самых простых заключается в использовании редактора командной строки.

Редакторы командной строки При использовании интерактивной оболочки Bourne Again Shell можно редактировать командную строку и команды из исторического файла, используя любой из имеющихся в оболочке Bourne Again Shell редакторов командной строки (vim или emacs). При использовании редактора командной строки работа, в отличие от автономного редактора vim, начинается в режиме ввода. Вы можете переключаться между командным режимом и режимом ввода. Редактор emacs не имеет режимов работы, но отделяет команды от вводимой информации путем распознавания в качестве команд управляющих символов.

Псевдонимы Псевдоним — это имя, которое оболочка переводит в другое имя или в сложную команду. Псевдонимы позволяют определять новые команды путем подстановки строки вместо первой лексемы в простой команде. Оболочки Bourne Again и TC для определения псевдонимов используют разные синтаксисы, но сами псевдонимы работают в обеих оболочках одинаково.

Функции Функция оболочки представляет собой серию команд, разбор которых, в отличие от команд сценария, производится еще до их сохранения в памяти. Благодаря этому функции оболочки запускаются быстрее сценариев оболочки. Сценарии оболочки разбираются в процессе выполнения и хранятся на диске. Функция может быть определена в командной строке или внутри сценария оболочки. Если нужно, чтобы определение функции оставалось действующим между сеансами входа в систему, эту функцию можно определить в файле запуска. Подобно функциям многих языков программирования, функция оболочки вызывается путем предоставления ее имени, за которым следуют аргументы.

Свойства оболочки Поведение оболочки может быть настроено несколькими способами. При вызове bash в командной строке можно использовать ключи. А для включения и выключения свойств можно использовать встроенные в bash команды set и shopt.

Расширение командной строки При обработке командной строки Bourne Again Shell может заменять некоторые слова расширенным текстом. Многие виды расширений командной строки вызываются наличием в слове специальных символов (например, лидирующего знака доллара, означающего переменную). Эти специальные символы перечислены в табл. 8.6. Расширения проводятся в определенном порядке. Следом за расширениями истории и псевдонима обычно проводятся расширения параметров и переменных, командные подстановки и расширения имен. Заключение слова в двойные кавычки подавляет все виды расширений, кроме рас-

ширений параметров и переменных. Одинарные кавычки подавляют все виды расширений, равно как и цитирование (отключение) специального символа путем установки перед ним обратного слэша.

Упражнения

1. Объясните следующий неожиданный результат:

```
$ whereis date
date: /bin/date …
$ echo $PATH
.:/usr/local/bin:/usr/bin:/bin
$ cat > date
echo "This is my own version of date."
$ ./date
Fri May 22 11:45:49 PDT 2009
```

2. Какими двумя способами можно выполнить сценарий оболочки при отсутствии прав на выполнение файла, в котором он содержится? Можно ли выполнить сценарий оболочки при отсутствии прав на чтение файла, в котором он содержится?
3. Для чего нужна переменная PATH?
   1) Установите значение переменной PATH, чтобы заставит оболочку вести поиск в следующих каталогах в указанном порядке:
      - /usr/local/bin
      - /usr/bin
      - /bin
      - /usr/kerberos/bin
      - Каталог bin в вашем личном каталоге.
      - Рабочий каталог.
   2) Если файл с именем doit имеется в каталоге /usr/bin и еще один файл с таким же именем имеется в вашем каталоге ~/bin, то какой из них будет выполнен? (Если предположить, что у вас есть право на выполнение обоих файлов.)
   3) Если значение переменной PATH не настроено на поиск в рабочем каталоге, то как можно выполнить программу, находящуюся в этом каталоге?
   4) Какой командой можно воспользоваться для добавления каталога /usr/games к концу списка каталогов в переменной PATH?
4. Предположим, произведено следующее присваивание:

```
$ person=zach
```

   Покажите, каким будет вывод следующих команд:
   1) echo $person
   2) echo '$person'
   3) echo "$person"
5. Следующий сценарий оболочки добавляет записи к файлу по имени journal-file, который находится в вашем личном каталоге. Этот сценарий помогает отслеживать телефонные переговоры и встречи.

```
$ cat journal
# journal: добавляет регистрационные записи в файл
# $HOME/journal-file
```

```
file=$HOME/journal-file
date >> $file
echo -n "Введите имя человека или группы: "
read name
echo "$name" >> $file
echo >> $file
cat >> $file
echo "-------------------------------------------------" >> $file
echo >> $file
```

1) Что нужно сделать со сценарием, чтобы его можно было выполнить?
2) Почему в первом случае приема выходящей информации в сценарии используется встроенная команда read, а во втором — утилита cat?

6. Предположим, что существуют каталоги /home/zach/grants/biblios и /home/zach/biblios. Назовите рабочий каталог пользователя Zach после выполнения каждой последовательности заданных команд. Объясните, что происходит в каждом из случаев.

1)
```
$ pwd
/home/zach/grants
$ CDPATH=$(pwd)
$ cd
$ cd biblios
```
2)
```
$ pwd
/home/zach/grants
$ CDPATH=$(pwd)
$ cd $HOME/biblios
```

7. Назовите два способа идентификации PID-номера процесса, в котором запущена оболочка входа в систему.
8. Введите следующую команду:
```
$ sleep 30 | cat /etc/inittab
```
Будет ли какой-нибудь вывод у команды sleep? Откуда утилита cat получит свои входные данные? Что должно случиться перед тем, как оболочка выведет свое следующее приглашение?

Упражнения повышенной сложности

1. Напишите последовательность команд или сценарий, демонстрирующий расширение переменной, происходящее перед расширением имени.
2. Напишите сценарий оболочки, который выводит имя выполняющей его оболочки.
3. Объясните поведение следующего сценария оболочки:
```
$ cat quote_demo
twoliner="Это строка 1.
Это строка 2."
echo "$twoliner"
echo $twoliner
```
1) Сколько аргументов видит каждая команда echo в этом сценарии? Обоснуйте ответ.
2) Переопределите значение переменной оболочки IFS, чтобы вывод второй команды echo был таким же, как и у первой команды.

12. Добавьте статус выхода предыдущей команды к вашему приглашению, чтобы это приглашение работало следующим образом:

```
$ [0] ls xxx
ls: xxx: No such file or directory  Такой каталог или файл отсутствует
$ [1]
```

13. Утилита dirname рассматривает свои аргументы как имена и записывает в стандартный вывод префикс имени, то есть все, что есть в полном имени, исключая последний компонент:

```
$ dirname a/b/c/d
a/b/c
```

Если дать утилите dirname в качестве аргумента простое имя файла (без символов /), dirname запишет в стандартный вывод .:

```
$ dirname simple
.
```

Реализуйте команду dirname в качестве функции оболочки bash. Сделайте так, чтобы она, при задании таких аргументов как /, вела себя соответствующим образом.

14. Реализуйте утилиту basename, которая записывает последний компонент полного имени своего аргументов в стандартный вывод, в виде функции оболочки bash. Например, если задано имя a/b/c/d, функция basename записывает d в стандартный вывод:

```
$ basename a/b/c/d
d
```

15. Имеющаяся в Linux утилита basename имеет необязательный второй аргумент. Если ввести команду basename *путь суффикс*, basename удаляет *суффикс* и префикс из *пути*:

```
$ basename src/shellfiles/prog.bash .bash
prog
$ basename src/shellfiles/prog.bash .c
prog.bash
```

Добавьте это свойство к функции, которая была написана для упражнения 14.

Глава 9. Оболочка TC Shell

Оболочка TC Shell (tcsh) выполняет те же функции, что и оболочка Bourne Again Shell и другие оболочки: она предоставляет интерфейс между пользователем и операционной системой Linux. TC Shell является интерактивным интерпретатором команд, а также высокоуровневым языком программирования. Хотя в каждый отдельно взятый момент времени используется только одна оболочка, если возникнет такая необходимость, можно будет свободно переключаться между оболочками. Вполне возможно, что в разных окнах нужно будет запустить разные оболочки. Неплохой основой для изучения данной главы могут послужить главы 8 и 10, так как приводимые в них сведения относятся не только к оболочке bash, но и к оболочке tcsh. В данной главе рассматриваются свойства tcsh, которых нет в bash и которые реализованы не так, как их двойники в bash. Главная страница, посвященная оболочке tcsh, находится по адресу www.tcsh.org.

Оболочка TC Shell является расширенной версией оболочки C Shell (csh), которая впервые появилась в операционной системе Berkeley UNIX. Буква «T» в названии TC Shell происходит от названий операционных систем TENEX и TOPS-20, от которых ей достались автозавершение команд и некоторые другие свойства. В tcsh имеется ряд свойств, отсутствующих в csh, включая автозавершение имен файлов и пользователей, редактирование командной строки и исправление орфографических ошибок. Так же как и csh, программа tcsh настроена на более терпимое отношение к ошибкам и на упрощенное использование. Путем правильной настройки переменных можно заставить tcsh предупреждать о предпосылках случайного выхода из системы или перезаписи файла. В настоящее время как в bash, так и в tcsh используются многие популярные свойства исходной оболочки C Shell.

Инструкция присваивания

Хотя многие функциональные возможности tcsh имеются и в bash, в синтаксисе некоторых команд наблюдается ряд отличий. Например, инструкция присваивания в tcsh имеет следующий синтаксис:

set переменная = значение

В bash пробелы по обе стороны от знака равенства не допустимы, а в tcsh они разрешены. По соглашению, переменные оболочки в tcsh имеют, как правило, имена, составленные из букв нижнего, а не верхнего регистра (но можно использовать оба регистра). Если сослаться на необъявленную переменную (на такую переменную, которой не присвоено значение), tcsh, в отличие от bash, выдаст сообщение об ошибке. И наконец, по умолчанию в tcsh в качестве приглашения используется знак «больше, чем» (>), но зачастую для приглашения устанавливается только знак доллара — $, за которым следует пробел. В данной главе в примерах используется приглашение в виде строки `tcsh $`, чтобы избежать путаницы с приглашением оболочки bash.

НЕ ИСПОЛЬЗУЙТЕ TCSH В КАЧЕСТВЕ ЯЗЫКА ПРОГРАММИРОВАНИЯ

Если есть опыт работы с UNIX и хорошие навыки работы с оболочками C или TC Shell, может появиться желание использовать tcsh в качестве оболочки для входа в систему. Но может оказаться, что оболочка TC Shell не настолько хороша в роли языка программирования, как оболочка bash. Если вы собираетесь изучать только один язык программирования сценариев, сделайте свой выбор в пользу bash. Оболочка Bourne Again Shell и оболочка dash (см. раздел «Оболочка dash» на с. 283), входящая в семейство bash, повсеместно используются в Linux для программирования многих сценариев, выполняющих задачи администрирования системы.

Сценарии оболочки

Оболочка TC Shell может выполнять файлы, содержащие команды tcsh, точно так же, как оболочка Bourne Again Shell может выполнять файлы, содержащие команды bash. Хотя в обеих оболочках применяются схожие концепции написания и выполнения сценариев, у них различаются методы объявления переменных и присваивания им значений, а также синтаксис управляющих структур.

Сценарии bash и tcsh можно запускать при использовании в качестве интерпретатора команд одной из этих оболочек. Для выбора оболочки, которая запустит сценарий, существует несколько методов. Дополнительные сведения даны в разделе «#! — способ указания оболочки» на с. 294.

Если первым символом сценария оболочки является знак решетки (#) и за ним не следует восклицательный знак (!), оболочка TC Shell выполняет сценарий под управлением tcsh. Если первым символом не является знак решетки, tcsh для выполнения сценария вызывает ссылку sh на dash или bash.

ECHO: ИЗБАВЛЕНИЕ ОТ КОДА КЛАВИШИ ВВОД

Для избавления от кода клавиши ВВОД, который имеющаяся в tcsh встроенная команда echo обычно выводит в конце строки, эта команда воспринимает либо ключ –n, либо замыкающие символы \c. Встроенная команда echo, принадлежащая оболочке bash, воспринимает только ключ –n (см. раздел «read: Получение пользовательского ввода» на с. 466).

ИГРЫ ОБОЛОЧЕК

Когда при работе с интерактивной оболочкой TC Shell вы запускаете сценарий, первым символом которого не является знак решетки (#), и вызываете сценарий непосредственно (не предваряя его имя словом tcsh), то для выполнения сценария оболочка tcsh вызывает ссылку sh на оболочку dash или bash. Следующий сценарий был написан для запуска под управлением tcsh, но при вызове из командной строки tcsh он выполняется оболочкой bash. Встроенная команда set (см. раздел «set: Инициализация аргументов командной строки» на с. 461) работает под управлением bash и tcsh по-разному. В результате следующий пример (из раздела «Чтение пользовательского ввода» на с. 390) выводит приглашение, но не ждет вашего ответа:

```
tcsh $ cat user_in
echo -n "Введите текст: "
set input_line = "$<"
echo $input_line
tcsh $ user_in
Введите текст:
```

Хотя, если пример запустить из следующей командной строки tcsh, где оболочка tcsh вызывается явным образом, сценарий выполняется под tcsh и работает корректно:

```
tcsh $ tcsh user_in
Введите текст: Это введенный текст
Это введенный текст
```

Вход в оболочку TC Shell и выход из нее

chsh

Вызвать выполнение программы tcsh можно путем ввода команды tcsh.

Чтобы развеять сомнения насчет типа используемой оболочки, следует ввести команду ps. Эта команда показывает, с какой оболочкой вы работаете: с tcsh, bash, sh (которая ссылается на bash) или с какой-нибудь другой. Команда finger, за которой следует имя пользователя, выводит имя оболочки, с помощью которой был осуществлен вход в систему; это имя хранится в файле /etc/passwd. (В Mac OS X вместо этого файла используется Open Directory.) Если вы намерены использовать tcsh, то для смены оболочки входа в систему следует воспользоваться командой chsh (change shell — сменить оболочку):

```
bash $ chsh
Changing shell for sam.        Смена оболочки для пользователя sam
Password:
New shell [/bin/bash]: /bin/tcsh
Shell changed.                 Смена оболочки произведена
bash $
```

Указанная вами оболочка будет запущена при следующем входе в систему и при всех последующих входах в систему, пока не будет указана другая оболочка для применения в этих целях. Имя оболочки, используемой для входа в систему, хранится в файле /etc/passwd.

Выйти из оболочки tcsh можно несколькими способами. Выбор способа обуславливается двумя факторами: установлено ли значение переменной оболочки ignoreeof и какая именно оболочка используется, та, которая применялась для входа в систему (своя оболочка для входа в систему), или другая оболочка, запущенная уже после входа в систему. При сомнениях в выборе способа выхода из tcsh следует нажать в строке CONTROL+D без пробелов в начале, как нужно завершить стандартный ввод в программу. При этом либо произойдет выход, либо будет выведена инструкция о том, как следует выйти из оболочки. Если переменная ignoreeof не установлена (см. далее раздел «ignoreeof» на с. 396) и она не была установлена для вас в файле запуска, то из любой оболочки можно выйти с помощью комбинации CONTROL+D (такую же процедуру можно применить и для выхода из оболочки Bourne Again Shell).

Ели переменная ignoreeof установлена, комбинация CONTROL+D не работает. Переменная ignoreeof заставляет оболочку вывести сообщение о том, как из нее можно выйти. Из tcsh всегда можно выйти с помощью команды exit. А команда logout позволяет выйти только из той оболочки, которая использовалась для входа в систему.

Файлы запуска

При входе в оболочку TC Shell она автоматически выполняет различные файлы запуска. Обычно эти файлы выполняются в том порядке, который описывается в данном разделе, но tcsh можно скомпилировать таким образом, чтобы она использовала другой порядок. Для выполнения команд, находящихся в файле, у вас должно быть право на чтение этого файла. Файлы запуска оболочки bash рассмотрены в разделе «Файлы запуска» на с. 284.

/etc/csh.cshrc и /etc/csh.login

Сначала оболочка выполняет команды, которые находятся в файлах /etc/csh.cshrc и /etc/csh.login. Пользователь, работающий с root-правами, может настроить эти файлы для установки характеристик по умолчанию для пользователей tcsh, и эти настройки будут распространяться на всю систему. В этих файлах содержится конфигурационная информация, действительная для всей системы, в том числе путь по умолчанию, место поиска почты и т. д.

.tcshrc и .cshrc

Затем оболочка ищет файл ~/.tcshrc или, если такого файла не существует, файл ~/.cshrc (~/ — краткая форма записи путевого имени вашего личного каталога; см. главу 4, раздел «Знак ~ (тильда) в полных именах» на с. 104). Эти файлы можно использовать для установки переменных и параметров, имеющих локальную область видимости для вашей оболочки. При каждом создании новой оболочки tcsh заново инициализирует для нее эти переменные. В следующем файле .tcshrc устанавливается ряд переменных оболочки, создаются два псевдонима (см. раздел «Псевдонимы» на с. 374) и добавляются два каталога в переменную path – один в начале списка и один в конце:

```
tcsh $ cat ~/.tcshrc
set noclobber
set dunique
set ignoreeof
set history=256
set path = (~/bin $path /usr/games)
alias h history
alias ll ls -l
```

.history

Оболочки, используемые для входа в систему, заново создают исторический список из содержимого файла ~/.history. Если существует переменная histfile, tcsh использует вместо файла .history файл, на который указывает эта переменная.

.login

Оболочки, используемые для входа в систему, считывают и выполняют команды из файла ~/.login. В этом файле содержатся команды, которые нужно выполнить только один раз, в начале каждого сеанса работы. Для объявления в нем переменных среды окружения (глобальных переменных) можно использовать команду setenv (см. раздел «Подстановка переменной» на с. 384). В вашем файле .login можно также объявить тип используемого терминала и установить некоторые его характеристики.

```
tcsh $ cat ~/.login
setenv history 200
setenv mail /var/spool/mail/$user
if ( -z $DISPLAY ) then
        setenv TERM vt100
    else
        setenv TERM xterm
endif
stty erase '^h' kill '^u' -lcase tab3
date '+Login on %A %B %d at %I:%M %p'
```

Предыдущий файл .login устанавливает тип используемого терминала путем присваивания значения переменной TERM (сначала инструкция if определяет, использу-

ется ли графический интерфейс, а затем, исходя из этого, устанавливает значение, которое должно быть присвоено переменной TERM). Затем запускается команда stty (см. часть V) для установки характеристик терминала и команда date (см. часть V) для вывода времени входа в систему.

/etc/csh.logout и .logout

При выходе из оболочки входа в систему оболочка TC Shell запускает последовательно файлы /etc/csh.logout и ~/.logout. Следующий образец файла .logout использует команду date для вывода времени выхода из системы. Команда sleep гарантирует, что у команды echo будет время для отображения сообщения перед тем, как система осуществит выход. Задержка может пригодиться для коммутируемых линий, требующих некоторого времени для отображения сообщения.

```
tcsh $ cat ~/.logout
date '+Logout on %A %B %d at %I:%M %p'
sleep 5
```

Свойства, общие для оболочек Bourne Again Shell и TC Shell

Большинство свойств, общих для обеих оболочек, bash и tcsh, унаследовано ими от исходной оболочки C Shell:

- Расширение командной строки (называемое также подстановкой; см. раздел «Расширение командной строки (подстановка)» на с. 371).
- История (см. раздел «История» на с. 372).
- Псевдонимы (см. раздел «Псевдонимы» на с. 374).
- Управление заданиями (см. раздел «Управление заданиями» на с. 376).
- Подстановка имен файлов (см. раздел «Подстановка имен файлов» на с. 376).
- Работа со стеком каталогов (см. раздел «Работа со стеком каталогов» на с. 377).
- Командная подстановка (см. раздел «Командная подстановка» на с. 377).

Подробное рассмотрение этих свойств дано в главе, посвященной bash. В данном разделе основное внимание уделено различиям в реализации этих свойств в bash и tcsh.

Расширение командной строки (подстановка)

Расширение командной строки в оболочке Bourne Again Shell представлено в разделе «Обработка командной строки» на с. 351. На man-странице, посвященной tcsh, вместо термина расширение — expansion, который применяется для оболочки bash, используется термин подстановка — substitution. Оболочка TC Shell использует следующий порядок сканирования каждой лексемы, проверяя возможности ее расширения на основе:

1) исторической подстановки;
2) подстановки псевдонима;
3) подстановки переменной;
4) командной подстановки;
5) подстановки имени файла;
6) подстановки стека каталогов.

История

Оболочка TC Shell каждой командной строке присваивает последовательный номер события. Этот номер события можно вывести как часть приглашения tcsh (см. раздел «prompt» на с. 392). Примеры в этом разделе показывают пронумерованные приглашения там, где они помогают проиллюстрировать поведение команды.

Встроенная команда history

Как и в оболочке bash, имеющаяся в tcsh встроенная команда history выводит события, присутствующие в вашем историческом списке. Список событий выводится так, что вверху показывается самое старое событие. Последнее событие исторического списка является этой самой командой history, которая выводит данный список. В следующем историческом списке, который ограничен десятью строками благодаря аргументу 10 команды history, команда 23 изменяет приглашение оболочки tcsh, чтобы в нем выводился номер исторического события. Справа от номера события появляется время выполнения каждой команды.

```
32 $ history 10
    23  23:59   set prompt = "! $ "
    24  23:59   ls -l
    25  23:59   cat temp
    26  0:00    rm temp
    27  0:00    vim memo
    28  0:00    lpr memo
    29  0:00    vim memo
    30  0:00    lpr memo
    31  0:00    rm memo
    32  0:00    history
```

Расширение истории

В обеих оболочках работают одни и те же указатели событий и слов. Например, группа символов !! в tcsh, точно так же, как и в bash, является ссылкой на предыдущее событие. Команда `!328` выполняет команду события номер 328; команда `!?txt?` выполняет самое последнее событие, в котором содержится строка `txt`. Дополнительные сведения даны в разделе «Использование восклицательного знака (!) для ссылки на событие» на с. 329. В табл. 9.1 перечислены несколько модификаторов слов, используемых в tcsh и отсутствующих в bash.

Таблица 9.1. Модификаторы слов

Модификатор	Функция
`u`	Превращение первой буквы из нижнего регистра в верхний
`l`	Превращение первой буквы из верхнего регистра в нижний
`a`	Применение следующего модификатора в глобальном режиме внутри отдельного слова

В команде можно использовать более одного модификатора слов. Например, модификатор `a` при использовании в комбинации с модификатором `u` или `l` позволяет изменять регистр букв во всем слове.

```
tcsh $ echo $VERSION
VERSION: Undefined variable.     Неопределенная переменная
tcsh $ echo !!:1:al
echo $version
tcsh 6.14.00 (Astron) 2005-03-25 (i486-intel-linux) options wide,nls,dl,...
```

Кроме использования в целях доступа к историческому списку указателя события, для доступа, модификации и выполнения предыдущей команды можно использовать редактор командной строки (см. раздел «Редактирование командной строки» на с. 381).

Переменные

Переменные, устанавливаемые для управления историческим списком в tcsh, отличаются от таких же переменных, используемых в bash. Там, где для определения количества событий, хранящихся в ходе сеанса и между сеансами, bash использует соответственно переменные HISTSIZE и HISTFILESIZE, tcsh использует для этих целей переменные history и savehist (табл. 9.2).

Таблица 9.2. Переменные истории

Переменная	Значение по умолчанию	Функция
history	100 событий	Максимальное количество событий, сохраняемых в ходе сеанса
histfile	~/.history	Местонахождение файла history
savehist	Не установлено	Максимальное количество событий, сохраняемых между сеансами

history и savehist

При выходе из оболочки tcsh самые последние выполненные команды сохраняются в вашем файле ~/.history. При следующем запуске оболочки с помощью этого файла инициализируется исторический список. Значение переменной savehist определяет количество строк, сохраняемых в файле .history (это значение не обязательно должно быть таким же, как у переменной history). Если значение переменной savehist не установлено, tcsh не сохраняет историческую информацию между сеансами. Переменные history и savehist должны быть локальными (то есть объявляться не с помощью setenv, а с помощью команды set). Переменная history содержит количество событий, запоминаемых в ходе сеанса работы, а переменная savehist содержит количество событий, запоминаемых между сеансами (см. табл. 9.2).

Если установить для переменной history слишком большое значение, она может использовать слишком большой объем памяти. Если значение этой переменой не установлено или установлено в нуль, оболочка не сохраняет команды. Чтобы установить объем исторического списка в 500 самых последних событий, нужно ввести следующую команду вручную или поместить ее в ваш файл запуска ~/.tcshrc:

```
tcsh $ set history = 500
```

Следующая команда заставляет tcsh сохранять между сеансами входа в систему 200 самых последних событий:

```
tcsh $ set savehist = 200
```

Эти два присваивания можно объединить в одну команду:

```
tcsh $ set history=500 savehist=200
```

После установки значения переменной savehist нужно выйти из системы и войти в нее снова; после возвращения в систему в вашем историческом списке появятся 200 самых последних событий. Если нужно поддерживать список событий от одного входа в систему до другого, следует установить значение переменной `savehist` в своем файле ~/.tcshrc.

histlit Если установить переменную `histlit` (history literal — буквальное представление истории), команда `history` выведет команды, сохраненные в историческом списке, в том самом виде, в котором они были набраны, без какой-либо интерпретации со стороны оболочки. В следующем примере показано действие этой переменной (сравните строки под номером 32):

```
tcsh $ cat /etc/csh.cshrc
…
tcsh $ cp !!:1 ~
cp /etc/csh.cshrc ~
tcsh $ set histlit
tcsh $ history
…
    31  9:35    cat /etc/csh.cshrc
    32  9:35    cp !!:1 ~
    33  9:35    set histlit
    34  9:35    history
tcsh $ unset histlit
tcsh $ history
…
    31  9:35    cat /etc/csh.cshrc
    32  9:35    cp /etc/csh.cshrc ~
    33  9:35    set histlit
    34  9:35    history
    35  9:35    unset histlit
    36  9:36    history
```

Оболочки bash и tcsh по-разному расширяют указатель исторических событий. Если ввести команду !250w, bash заменит ее командой под номером 250, с добавленным к ней символом w. В отличие от этого tcsh просмотрит историю на предмет выполнения события, которое начинается со строки 250w. Причина разного подхода заключается в следующем: bash интерпретирует первые три символа строки 250w как номер команды, а tcsh интерпретирует эти символы как часть строки поиска 250w. (Если строка 250 стоит обособленно, tcsh рассматривает ее как номер команды.)

Если нужно добавить символ w к команде номер 250, номер события можно изолировать от w, заключив его в фигурные скобки:

```
!{250}w
```

Псевдонимы

Встроенные команды `alias` — `unalias`, имеющиеся в tcsh, очень похожи на своих двойников, имеющихся в bash (см. раздел «Псевдонимы» на с. 341). Но для встроенной команды alias используется немного другой синтаксис:

alias имя значение

Следующая команда создает псевдоним для команды ls:

```
tcsh $ alias ls "ls -lF"
```

В отличие от bash псевдоним в tcsh позволяет осуществлять подстановку аргументов командной строки:

```
$ alias nam "echo Hello, \!^ is my name"
$ nam Sam
Hello, Sam is my name
```

Строка \!* внутри псевдонима расширяется во все аргументы командной строки:

```
$ alias sortprint "sort \!* | lpr"
```

Следующий псевдоним выводит свой второй аргумент:

```
$ alias n2 "echo \!:2"
```

Специальные псевдонимы

Некоторые имена псевдонимов, которые называются *специальными псевдонимами*, имеют для tcsh особое значение. Если определить псевдоним с одним из этих имен, tcsh его автоматически выполняет, в соответствии с объяснениями, которые даны в табл. 9.3. Изначально все специальные псевдонимы не определены.

Чтобы просмотреть список текущих псевдонимов, нужно ввести команду **alias**. Для просмотра псевдонима с определенным именем нужно ввести команду **alias**, за которой следует это имя.

Таблица 9.3. Специальные псевдонимы

Псевдоним	Условие выполнения
beepcmd	При каждом обычном вызове оболочкой звукового сигнала терминала. Предоставляет способ получения других визуальных или звуковых эффектов для подобных случаев
cwdcmd	При каждом случае изменения рабочего каталога
periodic	Периодически, согласно указанному в переменной tperiod количеству минут. Если значение для переменной tperiod не установлено, или равно 0, periodic не имеет никакого значения
precmd	Непосредственно перед тем, как оболочка выводит приглашение
shell	Выдает абсолютное путевое имя оболочки, которое требуется использовать для запуска сценариев, не начинающихся с символов #! (раздел «#! — способ указания оболочки» на с. 294)

Подстановка истории в псевдонимах

Аргументы командной строки можно подставлять путем использования механизма истории, где одинарный восклицательный знак представляет командную строку, содержащую псевдоним. При этом используются те же модификаторы, которые рассматривались для работы с историей в главе 8. В следующем примере восклицательный знак процитирован, поэтому оболочка не интерпретирует его при создании псевдонимов:

```
21 $ alias last echo \!:$
22 $ last this is just a test
test
23 $ alias fn2 echo \!:2:t
24 $ fn2 /home/sam/test /home/zach/temp /home/barbara/new
temp
```

Событие 21 определено для last, псевдонима, который выводит последний аргумент. Событие 23 определено для fn2, псевдонима, который выводит простое имя файла или последнюю часть второго аргумента командной строки.

Управление заданиями

Управление заданиями осуществляется в bash (см. раздел «Управление заданиями» на с. 299) и в tcsh одинаково. Команды можно переводить из фонового в активный режим выполнения и обратно, можно временно приостанавливать задания и выводить список текущих заданий. Когда за символом % следует номер задания или строковый префикс, идентифицирующий задание, это сочетание является ссылкой на задание. При запуске на выполнение в фоновом режиме командной строки, порождающей несколько процессов из каждой оболочки, будут заметны незначительные различия. Если bash выводит только PID-номер последнего фонового процесса в каждом задании, tcsh выводит номера всех процессов, принадлежащих заданию. Пример, приведенный в разделе «Управление заданиями» на с. 299, под управлением tcsh выглядит следующим образом:

```
tcsh $ find . -print | sort | lpr & grep -l zach /tmp/* > zachfiles &
[1] 18839 18840 18841
[2] 18876
```

Подстановка имени файла

Оболочка TC Shell расширяет символы *, ? и [ ] в путевых именах точно так же, как это делает bash (см. раздел «Генерация имени файла и расширение имени» на с. 157). Символ * соответствует строке из нуля и более символов, символ ? соответствует любому одному символу, символы [ ] определяют класс символов, который используется для соответствия одному из символов, стоящих между парой квадратных скобок.

Оболочка TC Shell расширяет в имена файлов аргументы командной строки, которые начинаются со знака тильды (~), во многом так же, как это делает оболочка bash (см. раздел «Автозавершение тильды» на с. 379). Символ ~ при этом обозначает ваш личный каталог или личный каталог того пользователя, чье имя за ним следует. Использующиеся в bash специальные расширения ~+ и ~– в tcsh не доступны.

В tcsh доступно расширение данных, заключенных в фигурные скобки (см. раздел «Расширение данных, заключенных в фигурные скобки» на с. 353). Как и расширение тильды, это расширение рассматривается как аспект подстановки имени файла, несмотря на то что оно может генерировать строки, не являющиеся настоящими именами файлов. В tcsh и в ее предшественнице csh процесс использования шаблонов для поиска соответствий именам файлов называется *подстановкой* (globbing), а сам шаблон называется *шаблоном подстановки*. Если tcsh не может идентифицировать один или несколько файлов, соответствующих шаблону подстановки, она выводит сообщение

об ошибке (если шаблон не содержит фигурные скобки). Установка переменной оболочки noglob подавляет подстановку имен файлов, как подстановку вместо тильды, так и интерпретацию содержимого, заключенного в фигурные скобки.

Работа со стеком каталогов

Работа со стеком каталогов, осуществляемая в tcsh, практически не отличается от такой же работы в bash (см. главу 8, раздел «Работа со стеком каталогов»). Встроенная команда dirs выводит содержимое стека, а встроенные команды pushd и popd помещают каталоги в стек и извлекают их из него.

Командная подстановка

В tcsh формат $(...), использующийся для командной подстановки, недоступен. Вместо него нужно использовать исходный формат `...`. Во всем остальном реализация этой подстановки в bash и в tcsh является одинаковой. Дополнительные сведения о командной подстановке даны в разделе «Командная подстановка» на с. 358.

Перенаправление стандартной ошибки

В tcsh как и в bash символ «больше, чем» (>) используется для перенаправления стандартного вывода, но запись 2> для перенаправления стандартной ошибки в tcsh не используется. При работе в tcsh для перенаправления стандартного вывода и стандартной ошибки нужно использовать этот символ в сочетании со следующим за ним символом амперсанда (>&). Хотя этот же условный знак можно использовать и в bash, но им пользуются крайне редко. В следующих примерах, как и в примерах, относящихся к bash и приведенных в разделе «Объединение стандартного вывода и стандартной ошибки» на с. 290, используется ссылка на несуществующий файл x и на файл y, содержащий всего одну строку:

```
tcsh $ cat x
cat: x: No such file or directory  Файл или каталог с таким именем не существует
tcsh $ cat y
This is y.                         Это y.
tcsh $ cat x y >& hold
tcsh $ cat hold
cat: x: No such file or directory  Файл или каталог с таким именем не существует
This is y.                         Это y.
```

При использовании в предыдущем примере аргумента y утилита cat отправляет строку на стандартный вывод. А использование x заставляет cat отправить сообщение об ошибке на стандартную ошибку.

В отличие от bash tcsh не предоставляет простой способ перенаправления стандартной ошибки отдельно от стандартного вывода. Разумное решение часто обеспечивается обходными путями. В следующем примере утилита cat запускается с аргументами x и y в подчиненной оболочке (круглые скобки обеспечивают запуск заключенной в них команды в подчиненной оболочке, см. раздел «( ) — средство группировки команд» на с. 298). Кроме этого внутри команды для подчиненной оболочки символ > пере-

направляет стандартный вывод в файл outfile. Вывод, отправляемый на стандартную ошибку, не касается подчиненной оболочки, а отправляется родительской оболочке, где он наряду со стандартным выводом отправляется в файл errfile. Поскольку стандартный вывод уже был перенаправлен, файл errfile содержит только вывод, отправленный на стандартную ошибку.

```
tcsh $ (cat x y > outfile) >& errfile
tcsh $ cat outfile
This is y.                          Это y.
tcsh $ cat errfile
cat: x: No such file or directory   Файл или каталог с таким именем не существует
```

Способ сочетания и перенаправления вывода может пригодиться в том случае, когда нужно выполнить медленно работающую команду в фоновом режиме и не хочется, чтобы ее вывод засорял экран терминала. Например, из-за того, что утилита find (см. часть V) требует для завершения своей работы некоторое время, будет лучше запустить ее в фоновом режиме. Следующая команда ищет в иерархии файловой системы все файлы, которые в своем имени содержат строку `biblio`. Эта команда запускается в фоновом режиме и отправляет свой вывод в файл findout. Поскольку утилита find отправляет на стандартную ошибку отчет о каталогах, на поиск в которых у вас отсутствуют права, файл findout содержит запись о всех найденных файлах, а также запись о каталогах, которые не могут быть просмотрены.

```
tcsh $ find / -name "*biblio*" -print >& findout &
```

Если в этом примере не объединить стандартную ошибку со стандартным выводом и перенаправить только стандартный вывод, сообщения об ошибках будут появляться на экране, а в файле findout будут перечислены только найденные файлы.

Пока команды, у которых их стандартный вывод перенаправлен в файл, работают в фоновом режиме, вы можете посмотреть на выведенную информацию, используя команду tail (см. часть V) с ключом `-f`. Этот ключ заставляет команду tail выводить новые строки по мере их записи в файл:

```
tcsh $ tail -f findout
```

Чтобы завершить выполнение команды tail, нужно нажать клавишу прерывания (обычно это CONTROL+C).

Работа с командной строкой

В этом разделе рассматривается автозавершение слов, редактирование командной строки и исправление орфографических ошибок.

Автозавершение слов

По вашему запросу оболочка TC Shell осуществляет автозавершение в командной строке имен файлов, команд и имен переменных. Для ссылки на все эти свойства в tcsh используется обобщающий термин *автозавершение слов*.

Автозавершение имен файлов

Оболочка TC Shell может автоматически завершить имя файла после того, как вы укажете его уникальный префикс. Автозавершение имени файла похоже на генерацию имени файла, но цель автозавершения — выбор отдельного файла. Вместе взятые эти возможности способствуют использованию длинных, описательных имен файлов.

Чтобы воспользоваться автозаверешением имени файла при его наборе в командной строке, нужно набрать ту часть имени, которой будет достаточно для уникальной идентификации файла в каталоге, и нажать клавишу TAB; tcsh заполнит имя и добавит к нему пробел, оставив курсор в позиции, позволяющей ввести дополнительные аргументы или нажать ВВОД. В следующем примере пользователь набирает команду `cat trig1A` и нажимает TAB; система заполняет всю оставшуюся часть имени файла, которое начинается с `trig1A`:

```
tcsh $ cat trig1A ▸ TAB ▸ cat trig1A.302488 ■
```

Если набранному вами префиксу соответствует два и более имен файлов, tcsh не может завершить имя файла без получения от вас дополнительной информации. Оболочка пытается максимально удлинить префикс, добавляя к нему по возможности символы, а затем выдает звуковой сигнал, чтобы указать на то, что для устранения неоднозначности ей нужно, чтобы вы набрали дополнительные символы:

```
tcsh $ ls h*
help.hist  help.trig01  help.txt
tcsh $ cat h ▸ TAB ▸ cat help. (сигнал)
```

Вы можете дополнить имя достаточным для устранения неоднозначности количеством символов и снова нажать клавишу TAB. Кроме этого вы можете нажать CONTROL+D, чтобы заставить tcsh вывести список соответствующих имен файлов:

```
tcsh $ cat help. ▸ CONTROL+D
help.hist    help.trig01  help.txt
tcsh $ cat help.■
```

После вывода имен файлов tcsh снова выводит командную строку, позволяя уточнить имя файла (и снова нажать TAB) или завершить набор имени файла вручную.

Автозавершение тильды

Оболочка TC Shell способна различить тильду (~), появляющуюся в качестве первого символа слова, и при нажатии клавиши TAB пытается расширить слово с тильдой в имя пользователя:

```
tcsh $ cd ~za ▸ TAB ▸ cd ~zach/■ ▸ ВВОД
tcsh $ pwd
/home/zach
```

Добавляя слэш (/), tcsh показывает, что завершенное слово является именем каталога. Слэш также упрощает продолжение набора путевого имени.

Автозавершение команд и переменных

Механизм, аналогичный тому, что использовался для выведения списка и автозавершения имен файлов, можно использовать при работе с командами и именами

переменных. Если не дано полное путевое имя, для попытки автозавершения имени команды оболочка использует значение переменной path. Варианты выбора, список которых выводит tcsh, могут находиться в разных каталогах.

```
tcsh $ up ▸ TAB (сигнал) ▸ CONTROL+D
up2date              updatedb             uptime
up2date-config       update-mime-database
up2date-nox          updmap
tcsh $ up ▸ t ▸ TAB ▸ uptime ■ ▸ ВВОД
9:59am up 31 days, 15:11, 7 users, load average: 0.03, 0.02, 0.00
```

Если, как в следующем примере, вы установите переменную autolist, оболочка автоматически выведет список вариантов, как только вы вызовете автозавершение, нажав клавишу TAB. Клавишу CONTROL+D нажимать не понадобится.

```
tcsh $ set autolist
tcsh $ up ▸ TAB (сигнал)
up2date              updatedb             uptime
up2date-config       update-mime-database
up2date-nox          updmap
tcsh $ up ▸ t ▸ TAB ▸ uptime ■ ▸ ВВОД
10:01am up 31 days, 15:14, 7 users, load average: 0.20, 0.06, 0.02
```

Если для переменной autolist установить значение ambiguous, то оболочка при нажатии клавиши TAB выведет список вариантов выбора только в том случае, если введенное слово является самым длинным префиксом набора команд. В противном случае нажатие TAB заставит оболочку добавить к слову один или несколько символов, пока оно не превратится в самый длинный префикс, а затем повторное нажатие TAB приведет к выводу списка вариантов:

```
tcsh $ set autolist=ambiguous
tcsh $ echo $h ▸ TAB (сигнал)
histfile history home
tcsh $ echo $h■ ▸ i ▸ TAB ▸ echo $hist■ ▸ TAB
histfile history
tcsh $ echo $hist■ ▸ o ▸ TAB ▸ echo $history ■ ▸ ВВОД
1000
```

Оболочка должна ориентироваться по контексту слова во вводимой строке, чтобы определить, чем оно является: именем файла, именем пользователя, командой или именем переменной. Первое слово во вводимой строке считается именем команды; если слово начинается со специального символа $, оно рассматривается как имя переменной и т. д. В следующем примере вторая команда which работает неправильно: контекст слова up делает его похожим на начало имени файла, а не на начало команды. Оболочка предоставляет команде which аргумент updates (не представляющий собой исполняемый файл), и команда which выводит сообщение об ошибке:

```
tcsh $ ls up*
updates
tcsh $ which updatedb ups uptime
/usr/bin/updatedb
/usr/local/bin/ups
/usr/bin/uptime
tcsh $ which up ▸ TAB ▸ which updates
updates: Command not found.    Команда не найдена
```

Редактирование командной строки

Bindkey

Свойства редактирования командной строки tcsh похожи на свойства, имеющиеся в оболочке bash. Можно воспользоваться либо режимом команд emacs (используется по умолчанию), либо режимом команд vi(m). Переключение в режим команд vi(m) осуществляется по команде `bindkey -v`, а переключение в режим команд emacs — по команде `bindkey -e`. Клавиши стрелок в обоих режимах привязаны ко вполне очевидным командам перемещений, позволяющих перемещаться вперед и назад (вверх и вниз) по историческому списку, а также вправо и влево по текущей командной строке.

Без аргументов встроенная команда bindkey выводит текущую карту привязки к командам редактора и клавишные последовательности, которые можно вводить с клавиатуры:

```
tcsh $ bindkey
Standard key bindings                          Стандартные привязки
"^@"            ->  set-mark-command
"^A"            ->  beginning-of-line
"^B"            ->  backward-char
"^C"            ->  tty-sigintr
"^D"            ->  delete-char-or-list-or-eof
…
Multi-character bindings                       Многосимвольные привязки
"^[[A"          -> up-history
"^[[B"          -> down-history
"^[[C"          -> forward-char
"^[[D"          -> backward-char
…
Arrow key bindings                             Привязки для клавиш-стрелок
down            -> down-history
up              -> up-history
left            -> backward-char
right           -> forward-char
home            -> beginning-of-line
end             -> end-of-line
```

Знак вставки ^ обозначает CONTROL (^B = CONTROL+B). Комбинация ^[обозначает META или ALT; в данном случае клавиша META или ALT нажимается и удерживается при нажатии клавиши для следующего указанного символа. Если эта подстановка не работает или если на используемой клавиатуре отсутствует клавиша META или клавиша ALT, нужно нажать и отпустить клавишу ESCAPE, а затем нажать клавишу для следующего указанного символа. Для сочетания ^[[F нужно нажать META+[или ALT+[, а после этого нажать клавишу F или нажать другую комбинацию — ESCAPE [F. Слова down-up-left-right обозначают клавиши стрелок соответственно вниз-вверх-влево-вправо, а слова home-end обозначают клвиши HOME и END на числовой клавиатуре. Дополнительные сведения о клавише META даны в главе 7, в разделе «Клавиши: система записи обозначений и использование» на с. 231.

Многие клавиатуры тех машин, которые работают под управлением OS X, не имеют клавиш META и ALT. Для использования команд, рассчитанных на их применение на машинах Macintosh, можно настроить клавишу OPTION.

В предыдущем примере показана информация, выведенная командой bindkey при работе пользователя в режиме emacs. Для вывода привязок клавиш в режиме vi(m) нужно перейти в этот режим (bindkey -v) и ввести команду bindkey еще раз. Чтобы облегчить чтение, вывод bindkey можно направить через канал команде less.

Исправление орфографических ошибок

Оболочку tcsh можно заставить исправлять орфографические ошибки в именах команд, файлов и переменных (но только при использовании привязки клавиш в стиле emacs). Исправление ошибок работает только в двух случаях: до нажатия клавиши ВВОД и после ее нажатия.

До нажатия клавиши ВВОД

При работе в tcsh для исправления слова или строки до нажатия клавиши ВВОД вы должны указать оболочке на то, что вы хотите получить от нее такую услугу. Для этого существует две функции: spell-line и spell-word:

```
$ bindkey | grep spell
"^[$"          -> spell-line
"^[S"          -> spell-word
"^[s"          -> spell-word
```

Информация, выводимая командой bindkey, показывает, что spell-line привязана к META+$ (ALT+$ или ESCAPE $), а spell-word привязана к META+S и META+s (ALT+s или ESCAPE s и ALT+S или ESCAPE S). Чтобы исправить написание слова, находящегося слева от курсора, следует нажать META+s. Нажатие META+$ вызывает функцию spell-line, которая пытается исправить все слова в командной строке:

```
tcsh $ ls
bigfile.gz
tcsh $ gunzipp ▸ META+s ▸ gunzip bigfele.gz ▸ META+s ▸ gunzip bigfile.gz
tcsh $ gunzip bigfele.gz ▸ META+$ ▸ gunzip bigfile.gz
tcsh $ ecno $usfr ▸ META+$ ▸ echo $user
```

После нажатия клавиши ВВОД

Переменная по имени correct управляет тем, что tcsh пытается исправить или завершить после нажатия клавиши ВВОД и до передачи командной строки вызываемой команде. Если не установить значение для переменной correct, tcsh ничего исправлять не станет:

```
tcsh $ unset correct
tcsh $ ls morning
morning
tcsh $ ecno $usfr morbing
usfr: Undefined variable.        Неопределенная переменная
```

Оболочка сообщает об ошибке в имени переменной, а не в имени команды, поскольку она расширяет переменную до выполнения команды (см. раздел «Расширение командной строки (подстановка)» на с. 371). Если ввести неверное имя команды без каких-либо аргументов, оболочка сообщит о неверном имени команды.

Установите значение переменной correct в cmd для исправления только команд, или в all для исправления команд, переменных и имен файлов, или в complete для завершения команд:

```
tcsh $ set correct = cmd
tcsh $ ecno $usfr morbing
CORRECT>echo $usfr morbing (y|n|e|a)? y
usfr: Undefined variable.      Неопределенная переменная
tcsh $ set correct = all
tcsh $ echo $usfr morbing

CORRECT>echo $user morning (y|n|e|a)? y
zach morning
```

Когда значение переменной correct установлено в cmd, tcsh исправляет имя команды с ecno на echo. Когда значение correct установлено в all, tcsh исправляет и имя команды, и имя переменной. Если в командной строке будет присутствовать имя файла, оболочка внесет исправления и в это имя.

Оболочка TC Shell выводит специальное приглашение, позволяющее вам ввести y, чтобы принять измененную командную строку, n, чтобы ее отклонить, e, чтобы ее отредактировать, или a, чтобы отменить команду. Специальное приглашение, используемое при исправлении орфографических ошибок, рассмотрено в разделе «prompt3».

В следующем примере после установки значения переменной correct пользователь допустил опечатку в имени команды ls, после чего оболочка tcsh вывела приглашение с предложением правильного имени команды. Поскольку команда, предложенная оболочкой в качестве замены, не была командой ls, пользователь выбрал редактирование командной строки. Оболочка оставила курсор сразу же за командой, чтобы пользователь смог исправить допущенную ошибку:

```
tcsh $ set correct=cmd
tcsh $ lx -l ▸ ВВОД (сигнал)
CORRECT>lex -l (y|n|e|a)? e
tcsh $ lx -l■
```

В случае присвоения переменной correct значения complete, tcsh пытается завершить имя команды тем же способом, что и при автозавершении имени файла (см. раздел «Автозавершение имена файлов» на с. 379). В следующем примере после установки значения переменной correct в complete пользователь вводит команду up. Оболочка отвечает сообщением о неоднозначной команде — Ambiguous command, поскольку с этих двух букв начинаются сразу несколько команд, у которых разные третьи буквы. Затем оболочка снова выводит командную строку. В этой ситуации пользователь может нажать TAB, чтобы получить список команд, начинающихся с up, но он решает ввести букву t и нажать клавишу ВВОД. Оболочка завершает команду, поскольку эти три буквы уникальным образом идентифицируют утилиту uptime:

```
tcsh $ set correct = complete
tcsh $ upВВОД
Ambiguous command                    Неоднозначная команда
tcsh $ up ▸ tВВОД ▸ uptime
4:45pm  up 5 days,  9:54,  5 users,  load average: 1.62, 0.83, 0.33
```

Переменные

Хотя tcsh сохраняет значения переменных в виде строки, вы можете работать с этими переменными и как с числовыми. Выражения в tcsh могут использовать арифметические, логические или условные операторы. Встроенная команда @ может вычислить целочисленное арифметическое выражение.

В данном разделе термин *числовая переменная* используется для описания строковой переменной, в которой содержится число, используемое tcsh в простом арифметическом или логическом арифметическом вычислении. Но на самом деле настоящие числовые переменные в tcsh отсутствуют.

Имя переменной

Имя переменной в tcsh может состоять из одного или нескольких, но не более чем из 20 символов, которые могут быть буквами, цифрами и знаками подчеркивания. Первый символ не может быть цифрой, но может быть знаком подчеркивания.

Подстановка переменной

Объявление переменных, их вывод и присваивание им значений осуществляется с помощью встроенных команд set, @ и setenv. Встроенные команды set и setenv предполагают работу с нечисловыми строковыми переменными. Встроенная команда @ работает только с числовыми переменными. Команды set и @ объявляют локальные переменные. Встроенная команда setenv объявляет переменную и помещает ее в среду вызовов всех дочерних процессов (делает ее глобальной). Использование команды setenv аналогично присваиванию переменной значения с последующим применением в отношении ее команды export в оболочке Bourne Again Shell. Локальные переменные и переменные среды окружения рассмотрены в разделе «Локальность переменных» на с. 454.

Как только переменная будет создана или просто будет объявлено о ее существовании, оболочка tcsh осуществляет подстановку значения этой переменной, когда ее имени в командной строке предшествует знак доллара ($). Если знак доллара будет процитирован путем установки перед ним обратного слэша или заключением его в одинарные кавычки, оболочка подстановку не осуществляет. Когда переменная заключена в двойные кавычки, подстановка осуществляется даже в том случае, если знак доллара процитирован с помощью предшествующего ему обратного слэша.

Строковые переменные

Оболочка TC Shell обрабатывает строковую переменную практически так же, как и оболочка Bourne Again Shell. Основное отличие заключается в ее объявлении и в присваивании ей значения: tcsh для этого использует вполне определенные команды, set (или setenv) для объявления и (или) присваивания значения строковой переменной.

```
tcsh $ set name = fred
tcsh $ echo $name
```

```
fred
tcsh $ set
argv    ()
cwd     /home/zach
home    /home/zach
name    fred
path    (/usr/local/bin /bin /usr/bin /usr/X11R6/bin)
prompt  $
shell   /bin/tcsh
status  0
term    vt100
user    zach
```

В первой строке примера объявляется переменная name и ей присваивается строка fred. В отличие от bash tcsh позволяет ставить пробелы вокруг знака равенства, но не требует их наличия. В следующей строке выводится значение переменной name. Когда команда set вводится без аргументов, она выводит список всех локальных переменных оболочки и их значения (ваш список будет длиннее приведенного в примере). Когда команда set вводится с именем переменной и без значения, то в качестве значения она устанавливает для этой переменной нулевую строку.

Для удаления переменной можно воспользоваться встроенной командой unset:

```
tcsh $ set name
tcsh $ echo $name

tcsh $ unset name
tcsh $ echo $name
name: Undefined variable.      · Неопределенная переменная
```

setenv

При использовании команды setenv имя переменной должно быть отделено от присваиваемой ей строки одним или несколькими пробелами, а знак равенства должен быть опущен. В следующем примере команда tcsh создает подчиненную оболочку, команда echo показывает, что переменная и ее значение известны подчиненной оболочке, а команда exit возвращает управление исходной оболочке. Проверьте, как работает этот пример, воспользовавшись вместо команды setenv командой set:

```
tcsh $ setenv SRCDIR /usr/local/src
tcsh $ tcsh
tcsh $ echo $SRCDIR
/usr/local/src
tcsh $ exit
```

Если команда setenv используется без аргументов, она выводит список переменных среды окружения (глобальных переменных), то есть тех переменных, которые передаются всем дочерним процессам оболочки. По соглашению, переменные среды окружения получают имена, состоящие из заглавных букв.

Как и при использовании команды set, ввод команды setenv с именем переменной, но без присваиваемого значения приводит к установке для переменной в качестве значения нулевой строки. Хотя для удаления локальных переменных и переменных среды окружения можно использовать команду unset, существует и команда unsetenv, которая может удалять только переменные среды окружения.

Массив строковых переменных

Массив является коллекцией строк, каждая из которых идентифицируется с помощью своего индекса (1, 2, 3 и т. д.). Массивы в tcsh используют индексацию, которая начинается с единицы (то есть первый элемент массива имеет индекс 1). Перед тем как получить возможность доступа к отдельному элементу массива, нужно объявить весь массив, присвоив значение каждому элементу. Список значений должен быть заключен в круглые скобки, и в качестве разделителя должен использоваться пробел:

```
8 $ set colors = (red green blue orange yellow)
9 $ echo $colors
red green blue orange yellow
10 $ echo $colors[3]
blue
11 $ echo $colors[2-4]
green blue orange
12 $ set shapes = ('' '' '' '' '')
13 $ echo $shapes

14 $ set shapes[4] = square
15 $ echo $shapes[4]
square
```

Событие 8 объявляет строковый массив по имени colors из пяти элементов и присваивает значение каждому из них. Если на момент объявления массива значения элементов неизвестны, его можно объявить содержащим необходимое количество нулевых элементов (событие 12).

Ссылаться на весь массив можно путем установки впереди его имени знака доллара (событие 9). Число в квадратных скобках, указанное после ссылки на массив, является ссылкой на элемент массива (события 10, 14 и 15). Два числа в квадратных скобках, разделенные дефисом, ссылаются на два и более смежных элементов массива (событие 11). Дополнительные сведения о массивах даны в разделе «Формы специальных переменных» на с. 389.

Числовые переменные

Встроенная команда @ присваивает результат численных расчетов числовой переменной (как описано в разделе «Переменные» на с. 373, в tcsh отсутствуют настоящие числовые переменные). С помощью команды @ можно объявлять отдельные числовые переменные, точно так же, как это делается командой set при объявлении нечисловых переменных. Но если команде @ дать нечисловой аргумент, будет выведено сообщение об ошибке. Точно так же как set, команда @ без аргументов выводит список всех переменных оболочки.

Многие выражения, которые в состоянии вычислять встроенная команда @ и распознаваемые этой командой операторы, взяты из языка программирования C. Следующий формат показывает объявление переменной или присваивание ей значения с помощью команды @ (пробел после @ является обязательным):

```
@ имя_переменной оператор выражение
```

Имя_переменной является именем той самой переменной, которой присваивается значение. *Оператор* — это один из операторов присваивания языка C: =, +=, -=, *=, /=

или %=. (Описание этих операторов см. в табл. 12.4.) *Выражение* — это арифметическое выражение, которое может включать большинство операторов C (см. следующий раздел). Внутри выражений могут использоваться параметры, поясняющие или изменяющие порядок вычислений. Части выражения, содержащие любые из следующих символов: <, >, & или |, могут быть взяты в круглые скобки.

ИСПОЛЬЗОВАТЬ ЗНАК $ ПРИ ПРИСВАИВАНИИ ЗНАЧЕНИЯ ПЕРЕМЕННОЙ НЕ СЛЕДУЕТ

Как и в bash, в tcsh в начале имен переменных, получающих присваиваемое значение (тех имен, которые стоят слева от оператора), не должно быть знака доллара ($). Поэтому

```
tcsh $ @ $answer = 5 + 5
```

приведет к следующему результату:

```
answer: Undefined variable.
              (Неопределенная переменная)
```

или, если переменная answer определена,

```
@: Variable name must begin with a letter.
      (Имя переменной должно начинаться с буквы)
```

в то же время команда

```
tcsh $ @ answer = 5 + 5
```

присвоит значение 10 переменной answer.

Выражения

Выражение может быть составлено из констант, переменных и большинства операторов bash (см. главу 10, раздел «Операторы»). Выражения, вычисление которых относится к файлам, а не к числовым переменным или строкам, рассмотрены в табл. 9.8.

Выражения подчиняются правилам:

1. Оболочка вычисляет отсутствующий или пустой аргумент как 0.
2. Все результаты являются десятичными числами.
3. Все остальные операторы, кроме != и ==, работают с числовыми аргументами.
4. Каждый элемент выражения нужно отделять от смежного с ним элемента пробелом, если только этот элемент не является символом &, |, <, >, (или).

Рассмотрим несколько примеров, в которых используется команда @:

```
216 $ @ count = 0
217 $ echo $count
0
218 $ @ count = ( 10 + 4 ) / 2
219 $ echo $count
7
220 $ @ result = ( $count < 5 )
221 $ echo $result
0
222 $ @ count += 5
223 $ echo $count
12
224 $ @ count++
225 $ echo $count
13
```

Событие 216 объявляет переменную count и присваивает ей значение 0. Событие 218 показывает результат арифметической операции, присвоенный переменной. Событие 220 использует команду @ для присваивания результата логической операции, в которой используется константа и переменная, переменной result. Значение операции является ложью — *false* (= 0), потому что переменная count не меньше 5. Событие 222 является сокращенной формой следующей инструкции присваивания:

```
tcsh $ @ count = $count + 5
```

Событие 224 использует постфиксный оператор для увеличения значения count на единицу.

Операторы последующего приращения и последующего уменьшения

Операторы последующего приращения (++) и последующего уменьшения (--), как показано в следующих примерах, могут использоваться только в тех выражениях, которые содержат одно имя переменной:

```
tcsh $ @ count = 0
tcsh $ @ count++
tcsh $ echo $count
1
tcsh $ @ next = $count++
@: Badly formed number.          Неверно сформированное число
```

В отличие от языка программирования C и оболочки bash, выражения в tcsh не могут использовать операторы предварительного приращения и предварительного уменьшения.

Массивы числовых переменных

Перед тем как воспользоваться командой @ для присваивания значений элементам массива числовых переменных, нужно использовать встроенную команду set, чтобы объявить этот массив. Встроенная команда set может присваивать любые значения элементам числового массива, включая нулевые, другие числовые значения и нулевые строки.

Присваивание значения элементу числового массива похоже на присваивание значения простой числовой переменной. Единственное отличие состоит в том, что элемент массива нужно указать или предоставить его индекс. Для этого используется следующий синтаксис:

```
@ имя_переменной[индекс] оператор выражение
```

Индекс определяет адресуемый элемент массива. Первый элемент имеет индекс 1. Индекс не может быть выражением, но может быть либо числовой константой, либо переменной. В предыдущем синтаксисе квадратные скобки, в которые заключен индекс, являются частью самого синтаксиса и не служат обозначением того, что индекс является необязательным элементом. Если указать слишком большой индекс для того массива, который объявлен с помощью команды set, tcsh выведет сообщение о том, что индекс выходит за пределы диапазона: @: Subscript out of range.

```
226 $ set ages = (0 0 0 0 0)
```

```
227 $ @ ages[2] = 15
228 $ @ ages[3] = ($ages[2] + 4)
229 $ echo $ages[3]
19
230 $ echo $ages
0 15 19 0 0
231 $ set index = 3
232 $ echo $ages[$index]
19
233 $ echo $ages[6]
ages: Subscript out of range.   Индекс выходит за пределы диапазона
```

Элементы числового массива ведут себя как простые числовые переменные. В событии 226 объявляется массив из пяти элементов, каждый из которых имеет значение 0. В событиях 227 и 228 присваиваются значения элементам массива, а в событии 229 выводится значение одного из элементов. В событии 230 выводятся все элементы массива, в событии 232 элемент указывается с помощью переменной, а в событии 233 демонстрируется вывод сообщения об ошибке выхода за пределы диапазона.

Фигурные скобки

Точно так же, как и bash, оболочка tcsh позволяет использовать фигурные скобки для выделения переменной из окружающего ее текста без использования разделителя:

```
$ set bb=abc
$ echo $bbdef
bbdef: Undefined variable.          Неопределенная переменная
$ echo ${bb}def
abcdef
```

Формы специальных переменных

Специальная переменная со следующим синтаксисом имеет в качестве значения количество элементов в массиве *имя_переменной*:

$#имя_переменной

Определить, была ли установлена переменная с *именем_переменной*, можно с помощью значения переменной, имеющей следующий синтаксис:

$?имя_переменной

Эта переменная имеет значение 1, если переменная с именем *имя_переменной* была установлена, и 0 в противном случае:

```
tcsh $ set days = (mon tues wed thurs fri)
tcsh $ echo $#days
5
tcsh $ echo $?days
1
tcsh $ unset days
tcsh $ echo $?days
0
```

Чтение пользовательского ввода

Чтобы внутри сценария оболочки tcsh прочитать строку с терминала и присвоить ее переменной, можно воспользоваться встроенной командой set. В следующем фрагменте сценарий оболочки приглашает пользователя ввести данные и читает введенную строку в переменную input_line:

```
echo -n "Введите данные: "
set input_line = "$<"
```

Значением переменной оболочки $< является строка из стандартного ввода. Кавычки вокруг переменной $< препятствуют тому, чтобы оболочка присвоила переменной input_line одно только первое слово строки ввода.

Переменные оболочки

Переменные оболочки TC Shell могут быть установлены самой оболочкой, унаследованы оболочкой у среды окружения или установлены пользователем и использованы оболочкой. Некоторые переменные содержат очень важные значения (например, PID-номер фонового процесса). Другие переменные работают как переключатели: включают функцию, если они объявлены, и выключают ее, если они не объявлены. Многие из переменных оболочки часто устанавливаются из одного из файлов запуска: ~/.login или ~/.tcshrc (см. раздел «Файлы запуска» на с. 369).

Переменные оболочки, принимающие значения

argv Содержит аргументы (позиционные параметры) той командной строки, из которой была вызвана оболочка. Как и все массивы tcsh, этот массив использует индексацию, начинающуюся с единицы; элемент argv[1] содержит первый аргумент командной строки. Ссылку на $argv[*n*] можно сократить до $*n*. Лексема argv[*] является ссылкой сразу на все аргументы; ее можно сократить до $*. На имя вызываемой программы можно сослаться с помощью $0 (см. раздел «Позиционные параметры» на с. 459). Оболочка Bourne Again Shell не использует форму argv, она использует только сокращенную форму. Присвоить значения элементам массива argv нельзя.

$#argv или $# Содержит количество элементов массива argv. См. раздел «Формы специальных переменных» на с. 389.

autolist Управляет автозавершением команд и переменных (см. раздел «Автозавершение команд и переменных» на с. 379).

autologout Позволяет tcsh автоматически выходить из системы при слишком длительном простое оболочки. Значением переменной является количество минут простоя, ожидаемое tcsh перед выходом из системы. По умолчанию устанавливается значение 60 минут, за исключением случаев запуска в графической среде, при которых эта переменная изначально не устанавливается.

cdpath Воздействует на работу команды cd точно так же, как в bash это делает переменная CDPATH (см. раздел «Переменная CDPATH: Расширение возможностей команды cd» на с. 317). Переменной cdpath присваивается значение

массива, состоящего из абсолютных путевых имен (см. ниже описание переменной path), и ее значение устанавливается, как правило, в файле ~/.login, в строке, имеющей следующий вид:

```
set cdpath = (/home/zach /home/zach/letters)
```

При вызове команды cd с простым именем файла она ведет поиск подкаталога с этим именем в рабочем каталоге. Если такой подкаталог не будет найден, cd ищет этот подкаталог в каталогах, перечисленных в переменной cdpath.

correct — Этой переменной присваивается значение cmd для автоматического исправления орфографических ошибок в именах команд, значение all для исправления всей командной строки и значение complete для автозавершения имен команд. Эта переменная воздействует на исправление, осуществляемое после нажатия клавиши ВВОД (см. раздел «После нажатия клавиши ВВОД» на с. 382).

cwd — В качестве значения этой переменной оболочка устанавливает имя рабочего каталога. Когда доступ к каталогу осуществляется через символьную ссылку (см. раздел «Символьные ссылки» на с. 130), tcsh присваивает переменной cwd имя символьной ссылки.

dirstack — Оболочка с помощью встроенных команд pushd, popd и dirs поддерживает в этой переменной стек каталогов. Дополнительные сведения даны в разделе «Работа со стеком каталогов» на с. 302.

fignore — Содержит массив суффиксов, которые оболочка tcsh игнорирует при автозавершении имен файлов.

gid — В качестве значения этой переменной оболочка устанавливает идентификатор вашей группы (group ID).

histfile — Содержит полное путевое имя файла, в котором исторический список хранится между сеансами входа в систему (см. подраздел «Переменные» в разделе «История» на с. 373). По умолчанию она имеет значение ~/.history.

history — Определяет размер исторического списка. См. раздел «История» на с. 372.

home или HOME — Содержит путевое имя личного каталога пользователя. На эту переменную встроенная команда cd ссылается при подстановке имени файла вместо тильды (~) (см. раздел «Расширение тильды» на с. 365).

mail — Указывает файлы и каталоги, разделенные пробелами, которые проверяются на наличие электронной почты. Оболочка TC Shell проверяет наличие новых почтовых сообщений каждые 10 минут, если первое слово в значении переменной mail является числом, то это число указывает в секундах периодичность проверки оболочкой наличия новых почтовых поступлений.

owd — В этой переменной оболочка хранит имя вашего прежнего (старого) рабочего каталога. Эта переменная эквивалентна переменной ~- оболочки bash.

path или PATH — Содержит список каталогов, в которых tcsh ищет исполняемые команды (см. раздел «Переменная PATH: Указание тех мест, где оболочка ищет программы» на с. 312). Если этот массив пуст или не установлен, команды мож-

но выполнять только при указании их полных путевых имен. Установить значение переменной path можно с помощью следующей команды:

```
tcsh $ set path = ( /usr/bin /bin /usr/local/bin /usr/bin/X11 ~/bin . )
```

prompt — Содержит первичное приглашение, аналогична переменной PS1, которая используется оболочкой bash (см. раздел «Переменная PS1: Приглашение пользователю (первичное)» на с. 314). Если эта переменная не установлена, в качестве приглашения используется символ > или, при работе с root-правами, символ #. Оболочка расширяет восклицательный знак в строке приглашения в текущий номер события. Типичная строка файла .tcshrc, устанавливающая значение переменной prompt, имеет следующий вид:

```
set prompt = '! $ '
```

В табл. 9.4 перечислены некоторые форматирующие последовательности, которые можно использовать в переменной prompt для достижения специальных эффектов.

Таблица 9.4. Форматирующие последовательности переменной prompt

Последовательность	Отображение в приглашении
%/	Значение переменной cwd (рабочий каталог)
%~	То же самое, что и при использовании %/, но заменяет путь к личному каталогу пользователя знаком тильды
%! или %h или !	Номер текущего события
%d	День недели
%D	День месяца
%m	Имя хоста без указания домена
%M	Полное имя хоста, включая домен
%n	Имя пользователя
%t	Время дня на данную минуту
%p	Время дня на данную секунду
%W	Месяц в формате мм
%y	Год в формате гг
%Y	Год в формате гггг
%#	Знак решетки (#), если пользователь работает с root-правами; в противном случае будет использоваться знак «больше чем» (>)
%?	Статус выхода из предыдущей команды

prompt2 — Содержит вторичное приглашение, которое tcsh использует для обозначения ожидания дополнительного ввода. По умолчанию имеет значение %R?. Когда оболочка TC Shell ожидает от вас завершения ввода команды, она заменяет символы %R нулевой строкой, при последовательном переборе в структуре foreach она заменяет эти символы словом foreach (см. раздел «Управляющая структура foreach» на с. 403), а при последовательном переборе в структуре while она заменяет эти символы словом while (см. раздел «Управляющая структура while» на с. 404).

Когда в середине взятой в кавычки строки, вводимой в командную строку, не ставится обратный слэш и нажимается клавиша ВВОД, оболочка tcsh выводит сообщение

об ошибке независимо от того, какие кавычки использовались, одинарные или двойные:

```
% echo "Please enter the three values
Unmatched ".                    Отсутствует закрывающая кавычка
```

В следующем примере символ, генерируемый первым нажатием клавиши ВВОД, процитирован (отключен), и оболочка интерпретирует его буквальным образом. При работе с tcsh одиночные или двойные кавычки приводят к одинаковому результату. В качестве приглашения используется вопросительный знак (?).

```
% echo "Пожалуйста, введите три значения, \
? необходимые для завершения транзакции."
Пожалуйста, введите три значения,
> необходимые для завершения транзакции.
```

prompt3 Содержит приглашение, используемое при автоматическом исправлении орфографических ошибок. По умолчанию этой переменной присваивается значение `CORRECT>%R (y|n|e|a)?`, где **R** заменяется исправленной строкой.

savehis Определяет количество команд, сохраняемых из исторического списка при выходе из системы. Эти события сохраняются в файле с именем ~/.history. Оболочка использует эти события как начальный исторический список при новом входе в систему, заставляя ваш исторический список продолжать свое существование от одного сеанса входа в систему к другому (см. раздел «history» на с. 373).

shell Содержит путевое имя используемой оболочки.

shlvl Содержит уровень оболочки. Оболочка TC Shell увеличивает значение этой переменной при каждом запуске подчиненной оболочки и уменьшает его после каждого выхода из подчиненной оболочки. Для оболочки входа в систему значение этой переменной устанавливается в 1.

status Содержит статус выхода, возвращенный последней командой. Аналогична переменной $?, используемой в оболочке bash (см. раздел «$?: Статус выхода» на с. 458).

tcsh Содержит номер версии запущенной оболочки tcsh.

time Обеспечивает две функции: автоматическое хронометрирование команд с использованием встроенной команды time и задание формата, используемого командой time. Этой переменной можно установить либо одиночное числовое значение, либо массив, содержащий числовое значение и строку. Числовое значение используется для управления автоматическим хронометрированием; для любой команды, выполнение которой заняло больше времени, чем количество отведенных на ее выполнение секунд процессорного времени, команда time выводит по окончании ее выполнения статистику команды. Значение 0 приводит к выводу статистики после выполнения каждой команды. Строка управляет форматированием статистики путем использования форматирующих последовательностей, включая и те последовательности, которые перечислены в табл. 9.5.

Таблица 9.5. Форматирующие последовательности, используемые в переменной time

Последовательность	Отображение
%U	Время, потраченное командой на выполнение пользовательского кода, в секундах процессорного времени (в режиме пользователя)
%S	Время, потраченное командой на выполнение системного кода, в секундах процессорного времени (в режиме ядра)
%E	Полное время (общее количество прошедшего времени), затраченное на команду
%P	Процентное отношение процессорного времени, затраченного на эту задачу в течение данного периода, вычисляется как (%U+%S)/%E
%W	Количество выгрузок командных процессов на диск
%X	Средний объем общей программной памяти, использованный командой, в килобайтах
%D	Средний объем памяти, отведенной под данные, использованный командой, в килобайтах
%K	Общий объем памяти, использованный командой (вычисляется как %X+%D), в килобайтах
%M	Максимальный объем памяти, использованный командой, в килобайтах
%F	Количество основных ошибок отсутствия страниц (количество страниц памяти, которые приходилось считывать с диска)
%I	Количество операций ввода
%O	Количество операций вывода

По умолчанию встроенная команда time использует строку

```
"%Uu %Ss %E %P%  %X+%Dk %I+%Oio %Fpf+%Ww"
```

которая генерирует вывод в следующем формате:

```
tcsh $ time
0.200u 0.340s 17:32:33.27 0.0%      0+0k 0+0io 1165pf+0w
```

Хронометрирование команд можно применять в случае возникновения озабоченности по поводу производительности системы. Если ваши команды постоянно показывают большое количество ошибок отсутствия страниц и большой объем свопирования, то у системы, скорее всего, недостаточен объем памяти и следует подумать о наращивании ее объема. Информацию, выводимую командой time, можно использовать для сравнения производительности различных системных конфигураций и программных алгоритмов.

tperiod Управляет частотой в минутах, с которой оболочка выполняет специальный псевдоним periodic (см. раздел «Специальные псевдонимы» на с. 375).

user Оболочка устанавливает в качестве значения этой переменной ваше имя пользователя.

version Оболочка устанавливает в качестве значения этой переменной подробную информацию о версии оболочки tcsh, выполняемой системой.

watch В качестве значения этой переменной устанавливается массив из пар, составленных из имен пользователей и типов терминалов для отсле-

живания входов в систему и выходов из нее. Слово any означает пользователя с любым именем или терминал любого типа, поэтому значение (any any) приводит к отслеживанию всех входов и выходов из системы на всех терминалах, а значение (zach ttyS1 any console $user any) приводит к отслеживанию для пользователя zach на терминале ttyS1 любого пользователя, обратившегося к системной консоли, и любых входов и выходов из системы с использованием вашей учетной записи (возможно, для отлова злоумышленников). По умолчанию входы в систему и выходы из нее проверяются каждые 10 минут, но это значение можно изменить, поставив в начале массива числовое значение, задающее количество минут между проверками. Если установить значение переменной watch в (1 any console), вход в систему и выход из нее любого пользователя будет контролироваться с периодичностью в одну минуту. Отчеты выводятся непосредственно перед выводом нового приглашения оболочки. Кроме того, немедленная проверка принудительно вызывается при любом выполнении встроенной команды log. Сведения о том, как можно управлять форматом сообщений, инициируемых значением переменной watch, даны ниже в описании переменной who.

who — Управляет форматом информации, выводимой в сообщениях, инициируемых значением переменной watch (табл. 9.6).

Таблица 9.6. Форматирующие последовательности, используемые в переменной who

Последовательность	Отображение
%n	Имя пользователя
%a	Действие, предпринятое пользователем
%l	Терминал, на котором предпринято действие
%M	Полное имя удаленного хоста (или локального, если нет удаленного), с которого было предпринято действие
$m	Имя хоста без доменного имени

Когда переменная who не установлена, для watch-сообщений используется следующая строка по умолчанию: "%n has %a %l from %m", которая генерирует следующую строку:

```
sam has logged on tty2 from local
```

$ — Как и в bash, эта переменная содержит PID-номер текущей оболочки; ее нужно использовать в формате $$.

Переменные оболочки, работающие как переключатели

Следующие переменные оболочки работают как переключатели, их значения не играют никакой роли. Если переменная была объявлена, оболочка предпринимает специальное действие. Если нет, действие не предпринимается или подавляется. Эти переменные можно установить в своем файле запуска ~/.tcshrc, в сценарии оболочки или из командной строки.

autocorrect — Заставляет оболочку предпринять попытку автоматического исправления орфографической ошибки непосредственно перед каждой попыткой автозавершения (см. раздел «Исправление орфографических ошибок» на с. 382).

dunique Обычно команда pushd просто помещает новый рабочий каталог на вершину стека каталогов, создавая прецедент появления в этом стеке множества продублированных записей. Установка переменной dunique заставляет оболочку искать и удалять любые записи, дублирующие ту запись, которая готова к помещению в стек.

echo Заставляет оболочку выводить каждую команду перед ее выполнением. Переменная echo устанавливается путем вызова tcsh с ключом -x или с помощью команды set.

filec Включает автозавершение имен файлов (см. раздел «Автозавершение имен файлов» на с. 379) при запуске tcsh как csh (и когда csh является ссылкой на tcsh).

histlit Включает вывод команд из исторического списка в том виде, в котором они были введены, без их интерпретации оболочкой (см. раздел «histlit» на с. 374).

ignoreeof Препятствует использованию комбинации CONTROL+D для выхода из оболочки, приводящему к случайному выходу из системы. При объявлении этой переменной для выхода из оболочки нужно использовать команды exit или logout.

listjobs Заставляет оболочку вывести список всех заданий при каждой приостановке задания.

listlinks Заставляет встроенную команду ls-F показать тип файла, на который указывает каждая символьная ссылка, а не помечать символьную ссылку символом @.

loginsh Устанавливается оболочкой, если текущая оболочка является оболочкой входа в систему.

nobeep Выключает все звуковые сигналы, подаваемые оболочкой.

noclobber Препятствует случайной перезаписи файла при перенаправлении вывода и препятствует созданию файла при попытке присоединения вывода к несуществующему файлу (табл. 9.7). Чтобы обойти действие noclobber, нужно добавить к символу, используемому для перенаправления или присоединения вывода, восклицательный знак (например, >! и >>!). Дополнительные сведения даны в разделе «noclobber: Предотвращение перезаписи файлов» на с. 148.

Таблица 9.7. Порядок работы переменной noclobber

Командная строка	noclobber не объявлена	noclobber объявлена
x > *файл*	Перенаправляет стандартный вывод из процесса *x* в *файл*. Переписывает содержимое *файла*, если он существует	Перенаправляет стандартный вывод из процесса *x* в *файл*. Если *файл* существует, оболочка выводит сообщение об ошибке и не перезаписывает файл
x >> *файл*	Перенаправляет стандартный вывод из процесса *x* в *файл*. Добавляет новый вывод к концу *файла*, если он существует. Создает *файл*, если он не существует	Перенаправляет стандартный вывод из процесса *x* в *файл*. Добавляет новый вывод к концу *файла*, если он существует. Оболочка выводит сообщение об ошибке, если *файл* не существует, и не создает *файл*

noglob Препятствует расширению оболочкой нестрогих ссылок на имена файлов. Допускает буквальное использование в командной строке или сценарии оболочки символов *, ?, ~ и [ ] без их цитирования.

nonomatch Заставляет оболочку передавать вызываемой команде нестрогие ссылки на имена файлов, не соответствующие именам файлов. Оболочка не расширяет ссылку на файлы. Если не устанавливать переменную nonomatch, оболочка tcsh генерирует сообщение об ошибке No match (Соответствий не найдено) и не выполняет команду.

```
tcsh $ cat questions?
cat: No match              Соответствий не найдено
tcsh $ set nonomatch
tcsh $ cat questions?
cat: questions?: No such file or directory
                           Файл или каталог с таким именем не существует
```

notify Когда эта переменная установлена, tcsh немедленно отправляет на экран сообщение о завершении выполнения фонового задания. Обычно tcsh оповещает о завершении выполнения задания непосредственно перед выводом следующего приглашения (см. раздел «Управление заданиями» на с. 376).

pushdtohome В результате установки этой переменной вызов команды pushd без аргументов приводит к смене каталогов на ваш личный каталог (что эквивалентно команде pushd -).

pushdsilent Заставляет команды pushd и popd не выводить стек каталогов.

rmstar Заставляет оболочку запрашивать подтверждение при вводе команды rm *.

verbose Заставляет оболочку выводить каждую команду после расширения истории (см. раздел «Расширение истории» на с. 372). Переменная устанавливается путем вызова tcsh с ключом -v или с помощью команды set.

visiblebell Заставляет заменить звуковые сигналы вспышкой экрана.

Управляющие структуры

В оболочке TC Shell используются преимущественно те же структуры, что и в оболочке Bourne Again Shell. Несмотря на некоторые различия в используемом синтаксисе, их действия остаются такими же. В этом разделе дается обзор различий между управляющими структурами, которые имеют место в этих двух оболочках. Дополнительные сведения даны в разделе «Управляющие структуры» на с. 417.

Управляющая структура if

Для управляющей структуры if используется следующий синтаксис

if (выражение)простая_команда

Управляющая структура if работает только с простыми командами и не работает с каналами или списками команд. Для выполнения более сложных команд следует использовать управляющую структуру if...then (см. ниже).

```
tcsh $ cat if_1
#!/bin/tcsh
# Процедура, демонстрирующая использование простой управляющей структуры if.
#
if ( $#argv == 0 ) echo "if_1: Аргументы отсутствуют."
```

Сценарий if_1 проверяет свой собственный запуск без аргументов. Если выражение, заключенное в круглые скобки, вычисляется в *true* — то есть если в командной строке не было аргументов, — структура if выводит сообщение.

Кроме логических выражений, примером которых может служить выражение, использовавшееся в сценарии if_1, можно использовать выражения, возвращающие значения на основе статуса файла. Для этого типа выражений используется следующий синтаксис:

-n имя_файла

где *n* — одно из значений, перечисленных в табл. 9.8.

Если результатом теста будет истинное значение (*true*), выражение будет иметь значение 1, а если результат будет ложным (*false*), значением выражения будет 0. Если указанный файл не существует или не доступен, tcsh вычисляет выражение в 0. В следующем примере выясняется, является ли указанный файл обычным файлом или файлом каталога (и не является ли он устройством или другим специальным файлом):

```
tcsh $ cat if_2
#!/bin/tcsh
if -f $1 echo "$1 является обычным файлом или файлом каталога."
```

Таблица 9.8. Значения оператора *n*

n	**Значение**
b	Файл является специальным блоковым файлом
c	Файл является специальным символьным файлом
d	Файл является файлом каталога
e	Файл существует
f	Файл является обычным файлом или файлом каталога
g	У файла есть набор битов, устанавливающих идентификатор группы (group ID)
k	У файла установлен бит флажка ограничения прав на удаление (sticky bit)
l	Файл является символьной ссылкой
o	Владельцем файла является пользователь
p	Файл является именованным каналом (FIFO)
r	У пользователя есть право на чтение файла
S	Файл является специальным файлом сокета
s	Файл имеет содержимое (его размер не является нулевым)
t	Описатель файла (однозначное число, заменяющее имя файла) открыт и подключен к экрану
u	У файла есть набор битов, устанавливающий идентификатор пользователя
w	У пользователя есть право на запись файла

n	**Значение**
X	Файл является либо встроенной командой, либо выполняемым файлом, найденным в результате поиска в каталогах, указанных в переменной $path
x	У пользователя есть право на выполнение файла
z	Файл имеет длину 0 байт

Если это имеет какой-нибудь смысл, то операторы можно комбинировать. Например, `-ox` *имя_файла* является истинным выражением, если вы являетесь владельцем файла и имеете право на его выполнение. Это выражение является эквивалентом выражения `-o` *имя_файла* `&& -x` *имя_файла*.

Некоторые операторы вместо истинного или ложного значения возвращают полезную информацию о файле. В них используется точно такой же формат *-n имя_файла*, где *n* — одно из значений, показанных в табл. 9.9.

Таблица 9.9. Значения оператора *n*

n	**Значение**
A	Последнее время доступа к файлу*
A:	Последнее время доступа к файлу, выведенное в удобном для чтения формате
M	Последнее время изменения файл*
M:	Последнее время изменения файла, выведенное в удобном для чтения формате
C	Последнее время изменения i-узла файла*
C:	Последнее время изменения i-узла файла, выведенное в удобном для чтения формате
D	Номер устройства для файла. Этот номер является уникальным идентификатором устройства (например, раздела диска), на котором помещается файл
I	Номер i-узла для файла. I-узел является уникальным идентификатором файла на конкретном устройстве
F	Строка формата *устройство:i-узел*. Эта строка является уникальным идентификатором файла в любом месте системы
N	Количество жестких ссылок на файл
P	Права доступа к файлу, показанные в восьмеричном формате без лидирующего нуля
U	Числовой идентификатор пользователя (user ID) владельца файла
U:	Имя пользователя, принадлежащее владельцу файла
G	Числовой идентификатор группы (group ID), имеющей права доступа к файлу
G:	Имя группы, имеющей права доступа к файлу
Z	Количество байтов в файле

*Время измеряется в секундах от *начала отсчета времени* (как правило, начало отсчета — 1 января 1970 года).

При каждой проверке можно использовать только один из этих операторов, и в последовательности, состоящей из нескольких операторов, он должен быть последним. Поскольку 0 (нуль) может быть вполне допустимым ответом, возвращаемым некоторыми из этих операторов (например, количество байтов в файле может

равняться 0), в большинстве случаев при отказе они возвращают –1, а не 0, возвращаемый в подобном случае логическими операторами. Единственным исключением является оператор F, который возвращает двоеточие, если не может определить устройство или i-узел для файла.

Если нужно воспользоваться одним из этих операторов вне выражения управляющей структуры, то для тестирования файла и вывода результата можно воспользоваться встроенной командой filetest:

```
tcsh $ filetest -z if_1
0
tcsh $ filetest -F if_1
2051:12694
tcsh $ filetest -Z if_1
131
```

Инструкция goto

Для инструкции goto используется следующий синтаксис:

goto метка

Встроенная команда goto передает управление инструкции, которая начинается со строки *метка:* Использование goto показано в следующем фрагменте сценария:

```
tcsh $ cat goto_1
#!/bin/tcsh
#
# тестирование двух аргументов
#
if ($#argv == 2) goto goodargs
echo "Порядок использования: $0 арг1 арг2"
exit 1
goodargs:
...
```

Сценарий goto_1 выводит сообщение о порядке использования (см. раздел «Сообщение о порядке использования» на с. 426), когда он вызывается с аргументами, количество которых больше или меньше двух.

Обработка прерывания

Инструкция onintr (on interrupt — при прерывании) осуществляет передачу управления в том случае, когда вы прерываете выполнение сценария оболочки. Для инструкции onintr используется следующий формат:

onintr метка

Когда в процессе выполнения сценария оболочки вы нажимаете клавишу прерывания, оболочка передает управление инструкции, которая начинается со строки *метка:*. Эта инструкция позволяет при прерывании выполнения завершить сценарий без возможных неприятностей. Ее можно использовать для того, чтобы при преры-

вании выполнения сценария оболочки и до возвращения управления родительской оболочке обеспечить удаление всех временных файлов.

Использование инструкции onintr показано в следующем сценарии. Пока вы не нажмете клавишу прерывания, этот сценарий находится в бесконечном цикле, а после нажатия этой клавиши он выводит сообщение и возвращает управление оболочке:

```
tcsh $ cat onintr_1
#!/bin/tcsh
# Демонстрация работы инструкции onintr
onintr close
while ( 1 )
    echo "Программа работает."
    sleep 2
end
close:
echo "Завершение программы."
```

Если сценарий создает временные файлы, то инструкцией onintr можно воспользоваться для их удаления.

```
close:
rm -f /tmp/$$*
```

Нестрогая ссылка на имена файлов /tmp/$$* соответствует всем файлам в каталоге /tmp, чьи имена начинаются с PID-номера текущей оболочки (см. описание этой технологии присвоения имен временным файлам в разделе «$$: PID-номер» на с. 457).

Управляющая структура if...then...else

Управляющая структура if...then...else имеет три формы. Первая форма, представляющая собой расширение простой структуры if, выполняет более сложные команды или группы команд, если выражение вычисляется в *true*. Эта форма по-прежнему не имеет никаких разветвлений.

```
if (выражение) then
    команды
endif
```

Вторая форма имеет разветвление по двум направлениям. Если выражение вычисляется в *true*, выполняется первый набор *команд*. Если оно вычисляется в *false*, выполняется второй набор команд, который следует за словом *else*.

```
if (выражение) then
    команды
else
    команды
endif
```

Третья форма аналогична структуре if...then...elif (см. раздел «Управляющая структура if...then...elif» на с. 423). Она выполняет тесты до тех пор, пока не найдет *выражение*, вычисляющееся в *true*, а затем выполняет соответствующие *команды*.

```
if (выражение) then
    команды
```

```
else if (выражение) then
    команды
...
else
    команды
endif
```

В следующей программе переменной class присваивается значение 0, 1, 2 или 3, в зависимости от значения первого аргумента командной строки. Чтобы программа лучше читалась, переменная class объявляется в самом начале, в чем она, собственно, до первого применения не нуждается. Кроме этого, в тех же целях значение первого аргумента командной строки присваивается переменной number.

```
tcsh $ cat if_else_1
#!/bin/tcsh
# процедура определения категории первого
# аргумента командной строки
set class
set number = $argv[1]
#
if ($number < 0) then
    @ class = 0
else if (0 <= $number && $number < 100) then
    @ class = 1
else if (100 <= $number && $number < 200) then
    @ class = 2
else
    @ class = 3
endif
#
echo "Номер $number относится к классу ${class}."
```

Первая инструкция if проверяет, не является ли значение переменной number меньше нуля. Если оно меньше нуля, сценарий присваивает переменной class значение 0 и передает управление инструкции, которая следует за ключевым словом endif. Если оно не меньше нуля, то вторая инструкция if проверяет, не находится ли значение переменной number в диапазоне от 0 до 100. Булев оператор И — && — выдает значение *true*, если выражения с каждой из его сторон вычисляются в *true*. Если значение переменной number находится в диапазоне от 0 до 100, переменной class присваивается значение 1 и управление передается инструкции, которая следует за ключевым словом endif. Похожий тест определяет, находится ли значение переменной number в диапазоне от 100 до 200. Если нет, то благодаря заключительному ключевому слову else переменной class присваивается значение 3. Закрывает управляющую структуру if ключевое слово endif. В последней инструкции для изоляции имени переменной class от следующей за ней точки используются фигурные скобки ({}). В данном случае изоляция от точки фигурными скобками осуществлена для большей ясности; оболочка не рассматривает знак пунктуации как часть имени переменной. Фигурные скобки понадобятся в том случае, когда нужно будет, чтобы сразу же после значения переменной следовали какие-нибудь другие символы.

Управляющая структура foreach

Структура foreach является аналогом используемой в оболочке bash структуры for... in (см. раздел «Управляющая структура for...in» на с. 429). Для нее используется следующий синтаксис:

```
foreach индекс_цикла (список_аргументов)
   команды
end
```

Эта структура занимается циклическим выполнением *команд*. При первом проходе цикла структура присваивает значение первого аргумента в *списке_аргументов* *индексу_цикла*. Когда управление будет передано инструкции end, оболочка присваивает *индексу_цикла* следующее значение из *списка_аргументов* и снова выполняет *команды*. Оболочка повторяет эту процедуру, пока не будут исчерпаны все аргументы из *списка_аргументов*.

В следующем сценарии оболочки tcsh структура foreach используется для циклической обработки тех файлов в рабочем каталоге, в именах которых содержится указанная строка символов, и изменяет эту строку. Вы можете, к примеру, использовать этот сценарий, чтобы заменить в именах файлов строку memo строкой letter. Благодаря этому имена фалов memo.1, dailymemo и memories превратятся соответственно в letter.1, dailyletter и letterries.

Этот сценарий требует использования двух аргументов: изменяемой (старой) строки и новой строки. В *списке_аргументов* структуры foreach используется нестрогая ссылка на имена файлов, применяемая для последовательного перебора всех файлов в рабочем каталоге, в именах которых содержится первый аргумент. Для каждого файла, чье имя соответствует нестрогой ссылке на имена файлов, утилита mv изменяет имя файла. Команды echo и sed появляются внутри знаков обратного апострофа (‘), которые являются индикаторами командной подстановки: выполнение команд, указанных внутри знаков обратного апострофа, приводит к замене этих апострофов и всего, что между ними находится. Дополнительные сведения даны в разделе «Командная подстановка» на с. 358. Утилита sed (см. главу 13, «Редактор sed») производит подстановку первого аргумента в имени файла вторым аргументом. Выражения $1 и $2 являются соответственно сокращенной формой выражений $argv[1] и $argv[2].

```
tcsh $ cat ren
#!/bin/tcsh
# Порядок использования: ren арг1 арг2
#                        заменяет строку арг1 в именах файлов
#                        в рабочем каталоге строкой арг2
if ($#argv != 2) goto usage
foreach i ( *$1* )
   mv $i 'echo $i | sed -n s/$1/$2/p'
end
exit 0

usage:
echo "Порядок использования: ren арг1 арг2"
exit 1
```

В следующем сценарии цикл **foreach** используется для присваивания аргументов командной строки элементам массива по имени `buffer`:

```
tcsh $ cat foreach_1
#!/bin/tcsh
# процедура заполнения до 20 нулевых элементов значениями аргументов argv
#
set buffer = (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)
set count = 1
#
if ($#argv > 20) goto toomany
#
foreach argument ($argv[*])
    set buffer[$count] = $argument
    @ count++
end
# ЗАМЕНИТЕ command В СЛЕДУЮЩЕЙ СТРОКЕ ИМЕНЕМ ТОЙ
# ПРОГРАММЫ, КОТОРУЮ ХОТИТЕ ВЫЗВАТЬ.
exec command $buffer[*]
#
toomany:
echo "Задано слишком много аргументов."
echo "Порядок использования: foreach_1 [до 20 аргументов]"
exit 1
```

Сценарий **foreach_1** вызывает еще одну программу, которая называется `command` с командной строкой, гарантированно содержащей 20 аргументов. Если сценарий **foreach_1** вызывается менее чем с двадцатью аргументами, он заполняет командную строку нулями, чтобы создать для программы `command` набор из 20 аргументов. Предоставление сценарию более 20 аргументов заставляет его вывести сообщение о порядке использования и выйти со статусом 1, обозначающим ошибку.

Структура **foreach** организует по одному запуску команд для каждого аргумента командной строки. При каждом проходе цикла **foreach** присваивает значение следующего аргумента командной строки переменной `argument`. Затем сценарий присваивает каждое из этих значений элементу массива `buffer`. Индекс массива `buffer` обслуживает переменная `count`. Постфиксный оператор обеспечивает приращение значения переменной `count`, используя встроенную команду @ (@ count++). Чтобы не инициировался новый процесс, программа `command` вызывается встроенной командой **exec** (которая имеется в **bash** и в **tcsh**; см. раздел «exec: Выполнение команды или перенаправление описателей файлов» на с. 469). Поскольку программа `command` уже вызвана, запустивший ее процесс больше не нужен, поэтому создавать новый процесс не требуется.

Управляющая структура while

Для управляющей структуры **while** используется следующий синтаксис:

```
while (выражение)
    команды
end
```

Эта структура продолжает циклическое выполнение *команд*, пока *выражение* вычисляется в *true*. Если в самый первый раз *выражение* вычисляется в *false*, структура *команд* не выполняет.

```
tcsh $ cat while_1
#!/bin/tcsh
# Демонстрация работы управляющей структуры while.
```

```
# Эта процедура суммирует числа от 1 и до n;
# n - первый аргумент командной строки.
#
set limit = $argv[1]
set index = 1
set sum = 0
#
while ($index <= $limit)
    @ sum += $index
    @ index++
end
#
echo "Сумма равна $sum"
```

Эта программа вычисляет сумму всех целых чисел до *n* включительно, где *n* — это первый аргумент командной строки. Оператор += присваивает значение выражения `sum + index` переменной `sum`.

Инструкции break и continue

Работу структур foreach или while можно прервать с помощью инструкции break или инструкции continue. Перед передачей управления, эти инструкции выполняют команды, оставшиеся в строке. Инструкция break передает управление той инструкции, которая находится после инструкции end, прекращая тем самым выполнение цикла. А инструкция continue передает управление инструкции end, которая продолжает выполнение цикла.

Управляющая структура switch

Управляющая структура switch является аналогом имеющейся в оболочке bash структуры case (см. раздел «Управляющая структура case» на с. 439):

```
switch (тестируемая_строка)

case шаблон:
    команды
breaksw

case шаблон:
    команды
breaksw
...
default:
    команды
breaksw

endsw
```

Инструкция breaksw передает управление той инструкции, которая следует за инструкцией endsw. Если инструкцию breaksw опустить, управление перейдет следующей команде. В *шаблоне* можно использовать любой специальный символ из списка, показанного в табл. 10.2, за исключением символа канала (|).

```
tcsh $ cat switch_1
#!/bin/tcsh
# Демонстрация работы управляющей стрктуры switch.
# Эта процедура тестирует первый аргумент командной строки
# на наличие слов yes или no в любой комбинации букв
# верхнего и нижнего регистров.
#
# Проверка существования argv[1]
if ($#argv != 1) then
    echo "Порядок использования: $0 [yes|no]"
    exit 1
else
    # Элемент argv[1] существует, настройка switch на его значение
    switch ($argv[1])
    # если YES
        case [yY][eE][sS]:
        echo "Первый аргумент имеет значение yes."
        breaksw
    #
    # если NO
        case [nN][oO]:
        echo " Первый аргумент имеет значение no."
        breaksw
    #
    # если условия не выполнены
        default:
        echo "Первый аргумент не имеет значения yes или no."
        breaksw
    endsw
endif
```

Встроенные команды

Встроенными командами (builtins) называются такие команды, которые являются частью оболочки (то есть встроены в нее). Когда в качестве команды вводится простое имя файла, оболочка сначала проверяет, не является ли оно именем встроенной команды. Если это имя встроенной команды, оболочка выполняет эту команду как часть вызывающего процесса; новый процесс для выполнения встроенной команды оболочка не ответвляет. Оболочке не нужно искать программы встроенных команд в структуре каталогов, поскольку она имеет к ним немедленный доступ.

Если простое имя файла, введенное в качестве команды, не является именем встроенной команды, оболочка ищет требуемую программу в структуре каталогов, руководствуясь при этом значением переменной PATH. Когда оболочка находит программу, для ее выполнения она ответвляет от себя новый процесс.

Несмотря на то что ключевые слова, использующиеся в управляющей структуре (if, foreach, endsw и т. д.), не перечислены в табл. 9.10, они являются встроенными командами. В табл. 9.10 дается описание многих встроенных команд tcsh, некоторые из них встроены также и в другие оболочки.

Таблица 9.10. Встроенные команды оболочки tcsh

Встроенная команда	Функция
% *задание*	Синоним встроенной команды fg. В качестве *задания* указывается номер задания, которое нужно перевести в активный режим (см. раздел «fg: Перевод задания в активный режим» на с. 300)
% *задание* &	Синоним встроенной команды bg. В качестве *задания* указывается номер задания, которое нужно перевести в фоновый режим (см. раздел «bg: Отправка задания в фоновый режим» на с. 301)
@	Аналог встроенной команды set, предназначенный для вычисления числовых выражений (см. раздел «Числовые переменные» на с. 386)
Alias	Создание и отображение псевдонимов; в bash используется синтаксис, отличный от синтаксиса tcsh (см. раздел «Псевдонимы» на с. 374)
Alloc	Вывод отчета о количестве свободной и используемой памяти
bg	Перемещение приостановленного задания в фоновый режим (см. раздел «bg: Отправка задания в фоновый режим» на с. 301)
bindkey	Управление отображением клавиш на команды редактора командной строки, используемого в tcs
bindkey	Без аргументов bindkey выводит список всех клавишных привязок (см. раздел «Редактирование командной строки» на с. 381)
bindkey -l	Вывод списка всех доступных команд редактирования с кратким описанием каждой команды
bindkey -e	Переключение редактора командной строки в режим команд emacs (см. раздел «Редактирование командной строки» на с. 381)
bindkey -v	Переключение редактора командной строки в режим команд vi(m) (см. раздел «Редактирование командной строки» на с. 381)
bindkey *клавиша команда*	Привязка команды редактора *команда* к клавише *клавиша*
bindkey -b *клавиша команда*	Действие, аналогичное предыдущему, но позволяющее указать управляющие клавиши путем использования формы *C–x* (где *x* — это символ, клавиша которого нажимается при удержании клавиши CONTROL), указать meta-последовательность клавиш как *M-x* (на большинстве клавиатур, используемых при работе с Linux, в качестве клавиши META используется клавиша ALT; в Mac OS X используется клавиша OPTION [чтобы ее использовать, нужно установить флажок option в утилите Terminal]) и указать функциональные клавиши как *F-x*
bindkey -c *клавиша команда*	Привязка клавиши *клавиша* к команде *команда*. Здесь *команда* является не командой редактора, а встроенной командой оболочки или исполняемой программой
bindkey -s *клавиша строка*	Принуждение tcsh к подстановке *строки* при нажатии *клавиш*
builtins	Вывод списка всех встроенных команд
cd или chdir	Осуществление смены рабочего каталога (см. раздел «cd: Смена рабочего каталога» на с. 108)
dirs	Вывод стека каталогов (см. раздел «dirs: Отображение стека» на с. 302)

продолжение ⇩

Таблица 9.10 (*продолжение*)

Встроенная команда	Функция
echo	Вывод собственных аргументов. Можно заставить echo не выводить в конце строки символ, генерируемый клавишей ВВОД, если использовать ключ -n (см. раздел «Чтение пользовательского ввода» на с. 390) или использовать в конце символы \c (см. раздел «read: Получение пользовательского ввода» на с. 466). Встроенная команда echo аналогична по своему назначению утилите echo (см. часть V)
eval	Сканирование и вычисление командной строки. Если поставить eval перед командой, то команда перед ее выполнением сканируется оболочкой дважды. Эта особенность может пригодиться для той команды, которая генерируется с помощью командной подстановки или подстановки переменной. Из-за порядка, в котором оболочка обрабатывает командную строку, для достижения желаемого результата иногда приходится повторять сканирование (см. раздел «Встроенная команда eval» на с. 346)
exec	Накладывание на текущую выполняемую программу другой программы, выполняемой в той же самой оболочке, при этом исходная программа утрачивается. Дополнительные сведения даны в разделе «exec: Выполнение команды или перенаправление описателей файлов» на с. 469, а также в строке «source» в данной таблице
exit	Выход из оболочки TC Shell. Когда за командой exit следует числовое значение, tcsh возвращает это число в качестве статуса выхода (см. раздел «$?: Статус выхода» на с. 458)
fg	Перевод задания в активный режим (см. раздел «fg: Перевод задания в активный режим» на с. 300)
Filetest	Получение одного из операторов запроса, за которым следует одно и более имен файлов, и применение оператора к каждому файлу, из тех, чьи имена указаны (см. табл. 9.9). Возвращение результата в виде списка с пробелами в качестве разделителей
Glob	Работа, аналогичная команде echo, но без отображения пробелов между ее аргументами и без сопровождения отображаемой информации разделителями строк
Hashstat	Сообщение об эффективности механизма хеширования tcsh. Этот механизм ускоряет процесс поиска по каталогам в вашем пути поиска, заданном в переменной path. См. также описания команд rehash и unhash в данной таблице
History	Отображение списка самых последних команд (см. раздел «Встроенная команда history» на с. 372)
Jobs	Вывод списка заданий (приостановленных команд и тех команд, которые выполняются в фоновом режиме)
Kill	Завершение задания или процесса (см. раздел «kill: Прерывание процесса» на с. 475)
Limit	Ограничение компьютерных ресурсов, которые могут использоваться текущим процессом, и любым создаваемым им процессом. Можно установить ограничение на количество секунд процессорного времени, отводимое на процесс, на размер файлов, создаваемых процессом и т. д.
Log	Немедленный вывод отчета, который обычно инициируется в tcsh каждые 10 минут установкой значения переменной оболочки watch (см. раздел «watch» на с. 394)

Встроенная команда	Функция
Login	Осуществление входа пользователя в систему. За ней может следовать имя пользователя
Logout	Завершение сеанса работы, если используется исходная оболочка (оболочка входа в систему)
ls-F	То же самое, что и команда `ls -F` (см. часть V), но работает быстрее. (Эта встроенная команда представляет собой последовательность символов ls–F, не имеющую пробелов.)
Nice	Понижает приоритет обработки команды или оболочки. Эта встроенная команда нужна в том случае, если вы хотите запустить команду, которая предъявляет большие требования к системе и чей вывод не нужен сиюминутно. При работе с root-правами команду nice можно использовать для подъема приоритета команды. Дополнительные сведения о встроенной команде nice и одноименной утилите даны в части V
Nohup	Предоставление возможностя выхода из системы без прекращения процессов, запущенных в фоновом режиме. Некоторые системы настраиваются на такое поведение по умолчанию. Сведения о встроенной команде nohup и одноименной утилите даны в части V
Notify	Принуждение оболочки к немедленному уведомлению при изменении статуса одного из ваших заданий (см. раздел «Управление заданиями» на с. 362)
onintr	Управление действием, которое вызывается прерыванием выполнения сценария (см. раздел «Обработка прерывания» на с. 400). Сведения об эквивалентной команде в оболочке bash даны в разделе «trap: Перехват сигнала» на с. 472
popd	Смена рабочего каталога на каталог, чье имя находится на вершине стека каталогов и удаление имени этого каталога из стека каталогов (см. раздел «Работа со стеком каталогов» на с. 302)
printenv	Вывод всех имен и значений переменных среды окружения
pushd	Смена рабочего каталога и помещение имени нового каталога на вершину стека каталогов (см. раздел «Работа со стеком каталогов» на с. 302)
rehash	Создание заново внутренних таблиц, использующих механизм хеширования. При вызове нового экземпляра tcsh механизм хеширования создает на основе значения переменной path отсортированный список всех доступных команд. Как только вы добавите команду к каталогу в переменной path, нужно воспользоваться командой rehash, чтобы создать заново отсортированный список команд. Если этого не сделать, tcsh не сможет найти новую команду (см. также описания команд hashstat и unhash в данной таблице)
repeat	Прием двух аргументов — счетчика и простой команды (без каналов и списка команд) — и множественный вызов команды согласно установке счетчика
sched	Выполнение команды в указанное время. Например, следующая команда заставляет оболочку вывести сообщение о необходимости посетить стоматолога в 10 часов утра: `tcsh $ sched 10:00 echo «Визит к стоматологу.»`. Без аргументов команда sched выводит список запланированных команд. Когда наступает время выполнения запланированной команды, tcsh выполняет команду до вывода приглашения

продолжение

Таблица 9.10 (*продолжение*)

Встроенная команда	Функция
set	Объявление, инициализация и вывод значений локальных переменных (см. раздел «Переменные»)
setenv	Объявление, инициализация и вывод значений переменных среды окружения (см. раздел «Переменные»)
shift	Аналог встроенной в оболочку bash команды shift (см. раздел «shift: Смещение аргументов командной строки» на с. 460). Без аргумента команда shift перемещает значения индексов массива argv. Для выполнения такой же операции над массивом ее можно использовать с аргументом, представляющим собой имя массива
source	Выполнение сценария оболочки, имя которого предоставлено этой команде в качестве аргумента: команда source не ответвляет еще один процесс. Аналогична имеющейся в оболочке bash встроенной команде . (точка) (см. раздел «Утилита . (точка) или source: Запуск файла запуска в текущей оболочке» на с. 287). Встроенная команда source ожидает имя сценария оболочки TC Shell, поэтому в сценарии не нужны начальные символы #!. Выполняет команду source текущая оболочка, поэтому сценарий может содержать такие команды, как set, которые воздействуют на текущую оболочку. После внесения изменений в свой файл .tcshrc или .login вы можете использовать команду source для выполнения ее из оболочки, придавая силу изменениям без выхода и повторного входа в систему. Встроенные команды source могут вкладываться друг в друга
stop	Остановка задания или процесса, запущенного в фоновом режиме. Встроенная команда stop может принимать сразу несколько аргументов
suspend	Остановка текущей оболочки и перевод ее выполнения в фоновый режим. Ее действие аналогично действию клавиши приостановки, которая останавливает задания, запущенные в активном режиме. Остановить оболочку входа в систему эта встроенная команда не может
time	Выполнение команды, заданной в качестве аргумента. Выводит итоговые сведения, связанные с показателями времени выполняемой команды, в соответствии со значением переменной оболочки time (см. подраздел «time» на с. 393). Без аргументов команда time выводит показатели характеристик, связанных со временем для текущей оболочки и ее дочерних оболочек
umask	Идентификация или изменение прав доступа, которые tcsh присваивает создаваемому вами файлу (см. часть V)
unalias	Удаление псевдонима (см. раздел «Псевдонимы» на с. 374)
unhash	Выключение механизма хеширования (см. также описания команд hashstat и rehash в данной таблице)
unlimit	Удаление ограничений (см. описание команды limit в данной таблице), наложенных на текущий процесс
unset	Удаление объявления переменной (см. раздел «Переменные»)
unsetenv	Удаление объявления переменной среды окружения (см. раздел «Переменные»

Встроенная команда	Функция
wait	Принуждение оболочки к ожиданию завершения всех дочерних процессов. Когда команда wait вводится в ответ на приглашение оболочки, tcsh не выводит приглашение, пока не завершат работу все процессы, выполняемые в фоновом режиме. Если прервать выполнение этой команды с помощью клавиши прерывания, то перед тем как оболочка выведет приглашение, команда wait выведет список фоновых процессов
where	Выявление всех мест нахождения команды, имя которой задано в качестве аргумента, и сообщение для каждой команды, что именно она собой представляет: псевдоним, встроенную команду или выполняемую программу, путь к которой указан в значении вашей переменной path
which	Аналог команды where, но сообщающий только о выполняемой команде, а не о всех разновидностях указанного аргумента. Эта встроенная команда работает значительно быстрее, чем Linux-утилита which, и сообщает о псевдонимах и встроенных командах

Заключение

Подобно Bourne Again Shell, оболочка TC Shell является не только интерпретатором команд, но и языком программирования. Оболочка TC Shell, в основе которой лежит оболочка C Shell, разработанная в Калифорнийском университете в Беркли, включает в себя такие популярные функции, как ведение истории, использование псевдонимов и управление заданиями.

При выборе командного интерпретатора можно отдать предпочтение оболочке tcsh, особенно если уже знакома оболочка C Shell. Чтобы поменять свою оболочку входа в систему на tcsh, можно воспользоваться командой chsh. Но запуск tcsh в качестве интерактивной оболочки не приводит к тому, что tcsh запускает сценарии оболочки; они могут по-прежнему запускаться оболочкой bash, пока в первой строке сценария не будет явным образом указана другая оболочка, или имя сценария не будет указано в качестве аргумента команды tcsh. Указание оболочки в первой строке гарантирует вполне ожидаемое поведение сценария.

При достаточном знакомстве с оболочкой bash можно заметить некоторые различия между этими двумя оболочками. Например, разный синтаксис, использующийся для присваивания значений переменным. Кроме того, оболочка tcsh допускает использование пробелов вокруг знака равенства. Для создания как числовых, так и нечисловых переменных и для присваивания им значений используется встроенная команда set. А встроенная команда @ может вычислять числовые выражения для их присваивания числовым переменным.

setenv Поскольку в tcsh отсутствует встроенная команда export, для создания переменных среды окружения (глобальных переменных) следует использовать встроенную команду setenv. С помощью команды setenv переменным можно также присваивать значения. Команда unset удаляет как локальные переменные, так и переменные среды окружения, а команда unsetenv удаляет только переменные среды окружения.

Псевдонимы Синтаксис имеющейся в tcsh встроенной команды alias несколько отличается от синтаксиса команды alias, имеющейся в оболочке bash.

В отличие от bash псевдонимы оболочки tcsh разрешают подстановку аргументов командной строки с использованием синтаксиса механизма истории.

Большинство других свойств tcsh, таких как история, автозавершение слов и редактирование командной строки, очень похожи на свои аналоги, имеющиеся в bash. Синтаксис имеющихся в tcsh управляющих структур немного отличается от управляющих структур bash, но по функциональности они эквивалентны друг другу.

Подстановка

Термин подстановка (globbing), унаследованный от исходной оболочки Bourne Shell, относится к соответствию строк, в которых содержатся специальные символы (* и ?), именам файлов. Если оболочка tcsh не может сгенерировать список имен файлов, соответствующих шаблону подстановки, она выводит сообщение об ошибке. В этом ее поведение отличается от поведения оболочки bash, которая просто оставляет шаблон без изменений.

В tcsh стандартная ошибка и стандартный вывод могут быть перенаправлены, но прямого способа их независимого перенаправления не существует. Для осуществления такого перенаправления требуется создание подчиненной оболочки, которая перенаправляет стандартный вывод в файл, оставляя стандартную ошибку доступной родительскому процессу.

Упражнения

1. Предположим, что вы работаете со следующим историческим списком:

```
37 mail zach
38 cd /home/sam/correspondence/business/cheese_co
39 less letter.0321
40 vim letter.0321
41 cp letter.0321 letter.0325
42 grep hansen letter.0325
43 vim letter.0325
44 lpr letter*
45 cd ../milk_co
46 pwd
47 vim wilson.0321 wilson.0329
```

   Используя механизм истории, дайте команды на:

   1) отправку почтового сообщения пользователю Zach;
   2) использование vim для редактирования файла по имени wilson.0329;
   3) отправку файла wilson.0329 на принтер;
   4) отправку на принтер файлов wilson.0321 и wilson.0329.

2. 1) Как вывести на экран список текущих действующих псевдонимов?
   2) Напишите псевдоним по имени homedots, который выводит список только тех скрытых файлов, которые имеются в вашем личном каталоге.
3. 1) Как помешать команде, запускаемой в фоновом режиме, отправлять вывод на терминал?
   2) Что можно сделать, если команда запущена в активном режиме, а позже вы решили, что ее нужно запустить в фоновом режиме?
4. Какую инструкцию можно поместить в ваш файл ~/.tcshrc для предотвращения случайной перезаписи файла во время перенаправления вывода? Как можно отменить данное свойство?

5. Предположим, что в рабочем каталоге имеются следующие файлы:

```
adams.ltr.03
adams.brief
adams.ltr.07
abelson.09
abelson.brief
anthony.073
anthony.brief
azevedo.99
```

Что произойдет, если нажать клавишу TAB после набора следующих команд?

1) less adams.l
2) cat a
3) ls ant
4) file az

Что произойдет, если нажать комбинацию CONTROL+D после набора следующих команд?

1) ls ab
2) less a

6. Напишите псевдоним по имени backup, который воспринимает в качестве аргументов имена файлов, и создает копии файлов с такими же именами и с расширениями .bak.
7. Напишите псевдоним по имени qmake (quiet make — молчаливая команда make), который запускает команду make со стандартным выводом и стандартной ошибкой, перенаправленными в файл по имени make.log. Команда qmake должна воспринимать те же ключи и аргументы, что и команда make.
8. Как заставить оболочку tcsh всегда выводить путевое имя рабочего каталога в составе ее приглашения?

Упражнения повышенной сложности

1. Какие строки нужно изменить в сценарии оболочки Bourne Again Shell command_menu (показанном в главе 10 после табл. 10.2), чтобы превратить его в сценарий оболочки TC Shell? Внесите изменения и проверьте работоспособность нового сценария.
2. Пользователи часто считают команду rm (даже в виде `rm -i`) слишком опасной, поскольку она безвозвратно удаляет файлы. Создайте псевдоним по имени delete, который перемещает файлы, указанные в качестве его аргумента или аргументов, в каталог ~/.trash. Создайте второй псевдоним по имени undelete, который перемещает файлы из каталога ~/.trash в рабочий каталог. Поместите в свой файл ~/.logout следующую строку, предназначенную для удаления любых файлов, которые были «удалены», то есть перемещены псевдонимом delete во время текущего сеанса работы:

```
/bin/rm -f $HOME/.trash/* >& /dev/null
```

Объясните, что может измениться, если вместо этой строки в файл ~/.logout будет помещена следующая строка:

```
rm $HOME/.trash/*
```

3. Внесите изменения в сценарий foreach_1 (приведенный в дополнительном материале в разделе «Управляющая структура foreach» на с. 403), позволяющие ему воспринимать в качестве аргумента команду, предназначенную для выполнения с помощью команды exec.

4. Перепишите программу while_1 (приведенную в разделе «Управляющая структура while» на с. 404), чтобы она работала быстрее. Для проверки сокращения времени выполнения воспользуйтесь встроенной командой time.
5. Напишите свою собственную версию команды find по имени myfind, которая будет записывать вывод в файл findout, но без засорения этого файла сообщениями об ошибках, которые генерируются, когда у вас нет прав на поиск в каталоге. Команда myfind должна воспринимать те же ключи и аргументы, что и команда find. Можете ли вы представить себе ситуацию, при которой команда myfind не будет работать так, как задумано?
6. Почему в том случае, если сценарию foreach_1 (приведенному в дополнительном материале в разделе «Управляющая структура foreach» на с. 403) предоставляется 20 или меньше аргументов, не выполняются команды, следующие за меткой toomany:? (Почему это происходит при отсутствии команды exit?)

ЧАСТЬ IV

Средства программирования

Глава 10. Программирование оболочки Bourne Again Shell . . 416
Глава 11. Язык сценариев Perl 504
Глава 12. Язык обработки шаблонов AWK 552
Глава 13. Редактор sed 585
Глава 14. Утилита безопасного копирования rsync 602

Глава 10. Программирование оболочки Bourne Again Shell

В главе 5 было дано общее представление об оболочках, а в главе 8 были рассмотрены подробности, касающиеся оболочки Bourne Again Shell. В этой главе рассматриваются дополнительные команды Bourne Again Shell, встроенные команды и другие понятия, выводящие вопросы программирования оболочки на должный уровень. Хотя системные администраторы могут извлечь из программирования оболочки вполне определенную пользу, для выполнения задач системного администрирования читать эту главу совсем не обязательно. Ее можно пропустить и прочитать позднее, когда появится желание.

В первой половине главы рассмотрены вопросы управляющих структур, которые также называются конструкциями алгоритма управления. Эти структуры позволяют создавать сценарии, осуществляющие циклический перебор аргументов командной строки, принимающие решения на основе значения переменной, составляющие меню и делающие много другое. Оболочка Bourne Again Shell использует такие же конструкции, которые встречаются в таких языках программирования высокого уровня, как C.

Далее рассматриваются параметры и переменные, дается подробный разбор переменных, представляющих собой массивы, проводится сравнение локальных и глобальных переменных, специальных и позиционных параметров. При исследовании встроенных команд рассматривается команда type, выводящая информацию о команде, и команда read, позволяющая сценарию оболочки принимать данные, вводимые пользователем. В разделе, посвященном встроенной команде exec, показывается порядок использования exec для эффективного выполнения команды путем замещения процесса и объясняется порядок использования exec для перенаправления ввода и вывода непосредственно из сценария.

Затем рассматривается встроенная команда trap, предоставляющая способ обнаружения сигналов операционной системы (например, сигнала, который генерируется при нажатии комбинации клавиш CONTROL+C) и ответа на них. Рассмотрение встроенных команд завершается командой kill, которая может прервать процесс, и командой getopts, которая упрощает для сценария оболочки разбор ключей. Некоторые, наиболее востребованные встроенные команды перечислены в табл. 10.6.

Затем рассматриваются арифметические и логические выражения, а также работающие с ними операторы. Заключительный раздел главы 10 посвящен конструкции и реализации двух довольно объемных сценариев оболочки.

В этой главе содержится множество примеров программ. Хотя они приведены для иллюстрации вполне определенных понятий, многие из них используют также информацию из предыдущих примеров. Такое совмещение не только закрепляет ваши общие знания, касающиеся программирования оболочек, но и демонстрирует возможности комбинирования команд для решения сложных задач. Выполнение примеров, приведенных в данной книге, внесение в них изменений и всяческое экспериментирование с ними — хороший способ для усвоения основных понятий.

НЕ НАЗЫВАЙТЕ СЦЕНАРИЙ ОБОЛОЧКИ ИМЕНЕМ TEST

Если дать сценарию оболочки имя test, может возникнуть непредвиденная проблема, поскольку такое же имя есть у утилиты Linux. В зависимости от значения переменной PATH и способа вызова программы можно запустить либо сценарий, либо утилиту, что приведет к нежелательным результатам.

В этой главе понятия проиллюстрированы с помощью простых примеров, за которыми в разделах с пометкой «Дополнительно» следуют более сложные примеры. С помощью этих дополнительных сценариев иллюстрируются традиционные практические задачи программирования оболочки и представляются некоторые утилиты Linux, часто используемые в сценариях. Эти разделы можно пропускать, не теряя при этом нити повествования. А затем, когда будут усвоены основные понятия, к ним можно вернуться.

Управляющие структуры

Команды, определяющие алгоритм управления, изменяют порядок выполнения команд, имеющихся в сценарии оболочки. В оболочке TC Shell (см. раздел «Управляющие структуры» на с. 397) для этих команд используется несколько иной синтаксис по сравнению с тем, который используется в оболочке Bourne Again Shell. Управляющие структуры включают в себя инструкции if...then, for...in, while, until и case. В дополнение, для изменения порядка выполнения команд в сценарии в связке с управляющими структурами работают инструкции break и continue.

Управляющая структура if...then

Для управляющей структуры if...then используется следующий синтаксис:

if ***тест-команда***
then
команды
fi

Слова, выделенные в описании синтаксиса ***жирным*** шрифтом, являются элементами, которые предоставляются вами, чтобы заставить структуру осуществить желаемые действия. Слова, не выделенные жирным шрифтом, являются ключевыми словами оболочки, используемыми для идентификации управляющей структуры.

Встроенная команда test

На рис. 10.1 показано, что инструкция if тестирует статус, возвращенный *тест-командой*, и на основе этого статуса передает управление. Конец структуры if обозначен инструкцией fi (if в обратном написании). В следующем сценарии выводится приглашение на ввод

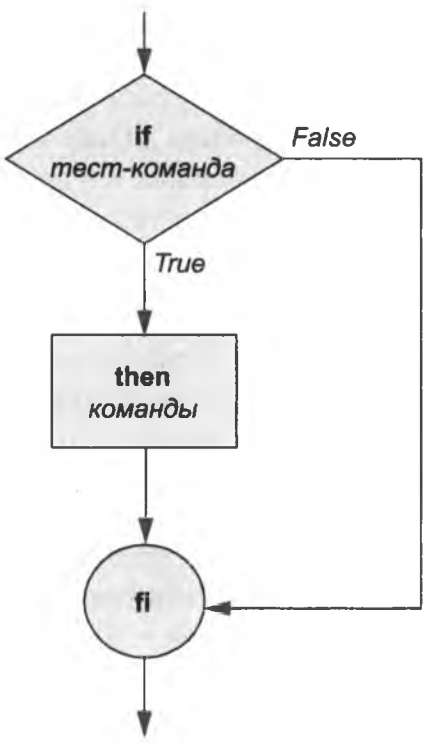

Рис. 10.1. Блок-схема управляющей структуры if..then

двух слов, производится их считывание, а затем для выполнения команд на основе результата, возвращенного встроенной командой test (в оболочке tcsh используется утилита test) после сравнения этих двух слов, используется структура if. (Сведения об утилите test, аналогичной встроенной команде test, даны в части V.) Встроенная команда test возвращает статус *true*, если оба слова одинаковы, и статус *false*, если они разные. Двойные кавычки вокруг имен переменных $word1 и $word2 гарантируют правильную работу команды test в том случае, если введена строка, содержащая пробелы или другие специальные символы:

```
$ cat if1
echo -n "слово 1: "
read word1
echo -n "слово 2: "
read word2

if test "$word1" = "$word2"
    then
        echo "Совпадают"
fi

echo "Конец программы."
$ ./if1
слово 1: peach
слово 2: peach
Совпадают
Конец программы.
```

В предыдущем примере в качестве *тест-команды* используется выражение test "$word1" = "$word2". Встроенная команда test возвращает статус *true*, если ее первый и третий аргументы имеют связь, на которую указывает ее второй аргумент. Если эта команда возвращает статус *true* (= 0), оболочка выполняет *команды*, находящиеся между инструкциями then и fi. Если команда возвращает статус *false* (не = 0), оболочка передает управление инструкции, которая следует за инструкцией fi, не выполняя инструкции, которые находятся между then и fi. Действие этой инструкции if заключается в выводе сообщения Совпадают, если два слова одинаковы. И в любом случае сценарий выводит сообщение Конец программы..

Встроенные команды (Builtins)

В оболочке Bourne Again Shell test является встроенной командой, то есть частью самой оболочки. Но это также и отдельная утилита, хранящаяся в каталоге /usr/bin/test. В этой главе рассматриваются и демонстрируются многие встроенные команды оболочки Bourne Again Shell. Каждая встроенная команда bash может иметь, а может и не иметь свой аналог в оболочке tcsh. Обычно если имеется встроенная версия команды, то она и используется, а если ее нет — используется утилита. Каждая версия команды от оболочки к оболочке может немного отличаться, или утилита может отличаться от любой из встроенных в оболочки команд.

Проверка аргументов

В следующей программе структура if используется в начале сценария для подтверждения передачи сценарию в командной строке хотя бы

одного аргумента. Используемый в команде **test** оператор `-eq` сравнивает два целых числа; специальный параметр **$#** (см. раздел «$#: Количество аргументов командной строки» на с. 459) принимает значение, равное количеству аргументов командной строки. Эта структура выводит сообщение и осуществляет выход из сценария со статусом выхода 1, если вами не предоставлен хотя бы один аргумент:

```
$ cat chkargs
if test $# -eq 0
   then
      echo "Вы должны предоставить хотя бы один аргумент."
      exit 1
fi
echo "Программа запущена."
$ ./chkargs
Вы должны предоставить хотя бы один аргумент.
$ ./chkargs abc
Программа запущена.
```

Тест, подобный показанному в сценарии **chkargs**, является ключевым компонентом любого сценария, для которого требуются аргументы. Чтобы избавить пользователя от получения бессмысленной или запутанной информации, сценарий должен проверять, предоставил ли пользователь соответствующие аргументы. Некоторые сценарии просто проверяют наличие аргументов (например, сценарий **chkargs**). Другие сценарии производят проверку на необходимое количество аргументов или конкретные их разновидности.

Можно провести тест на статус файлового аргумента или на связь между двумя файловыми аргументами. После проверки ввода в командную строку хотя бы одного аргумента следующий сценарий проводит тест на то, что этот аргумент является именем обычного файла (не являющегося каталогом или ссылкой) в рабочем каталоге. Проверка файла осуществляется с помощью встроенной команды **test** с ключом `-f` и первого аргумента командной строки (`$1`):

```
$ cat is_ordfile
if test $# -eq 0
   then
      echo "Вы должны предоставить хотя бы один аргумент."
      exit 1
fi
if test -f "$1"
   then
      echo "$1 обычный файл в рабочем каталоге"
   else
      echo "$1 НЕ ЯВЛЯЕТСЯ обычным файлом в рабочем каталоге "
fi
```

Используя ключи команды **test**, показанные в табл. 10.1, можно протестировать множество других характеристик файла.

Другие ключи команды **test** предоставляют способы тестирования связей между двумя файлами, например новее один файл другого или нет. Дополнительные сведения даны в следующих примерах этой главы и в описании утилиты **test** в части V.

Таблица 10.1. Ключи встроенной команды test

Ключи	Тестирование файла на предмет
-d	существования и отношения к файлу каталога
-e	существования
-f	существования и отношения к обычному файлу (не к файлу каталога)
-r	существования и наличия у вас права на чтение этого файла
-s	существования и ненулевого размера (более 0 байтов)
-w	существования и наличия у вас права на запись в этот файл
-x	существование и наличие у вас права на выполнение этого файла

ВСЕГДА ПРОВОДИТЕ ТЕСТИРОВАНИЕ АРГУМЕНТОВ

Для сокращения приводимых в этой книге примеров и для концентрации внимания на конкретных понятиях код, предназначенный для проверки аргументов, зачастую опускается или дается в сокращенном виде. Тестирование аргументов в программах оболочки, предназначенных для использования другими людьми, является правилом хорошего тона. Реализация такого тестирования упрощает работу и отладку сценариев.

[ ] — синоним команды test

В следующем примере, который является еще одной версией сценария chkargs, проверка аргументов проводится более традиционным для сценариев оболочки Linux способом. В этом примере в качестве синонима команды test используются квадратные скобки ([ ]). Вместо использования в сценариях слова test можно заключить тестируемые аргументы в квадратные скобки. Вокруг квадратных скобок должны быть пробельные символы (пробелы или символы табуляции).

```
$ cat chkargs2
if [ $# -eq 0 ]
    then
        echo "Порядок использования: chkargs2 аргумент…" 1>&2
        exit 1
fi
echo "Программа запущена."
exit 0
$ ./chkargs2
Порядок использования: chkargs2 аргумент…
$ ./chkargs2 abc
Программа запущена.
```

Сообщения о порядке использования

Сообщение об ошибке, которое выводит сценарий chkargs2, называется *сообщением о порядке использования* и использует систему записи 1>&2 для перенаправления ее вывода на стандартную ошибку (см. раздел «Перенаправление стандартной ошибки» на с. 288). После выдачи сообщения о порядке использования сценарий chkargs2 осуществляет выход со статусом 1, который свидетельствует о возникновении ошибки. Команда exit 0, находящаяся в самом конце сценария, заставляет chkargs2 выйти со статусом 0 после того, как программа запустится без ошибки. Если код возвращения статуса опустить, то оболочка Bourne Again Shell возвращает статус 0.

Сообщение о порядке использования обычно применяется для указания типа и количества аргументов, воспринимаемых сценарием. Сообщения о порядке исполь-

зования предоставляются многими утилитами Linux, полностью аналогично тому, как это делается в сценарии chkargs2. Если вызвать утилиту или другую программу с неверным количеством или типом аргументов, то они, чаще всего, выводят сообщение о порядке использования. В следующем примере показывается сообщение о порядке использования, выводимое командой cp, при вызове ее без аргументов:

```
$ cp
cp: missing file operand                    Отсутствует файловый операнд
Try 'cp --help' for more information.       Для получения дополнительных сведений
                                            введите команду 'cp --help'
```

Управляющая структура if...then...else

Введение инструкции else превращает инструкцию if в разветвление на два маршрута, которое показано на рис. 10.2. Для управляющей структуры if...then...else (доступной в tcsh, но немного с другим синтаксисом) используется следующий синтаксис:

```
if тест-команда
    then
        команды
    else
        команды
fi
```

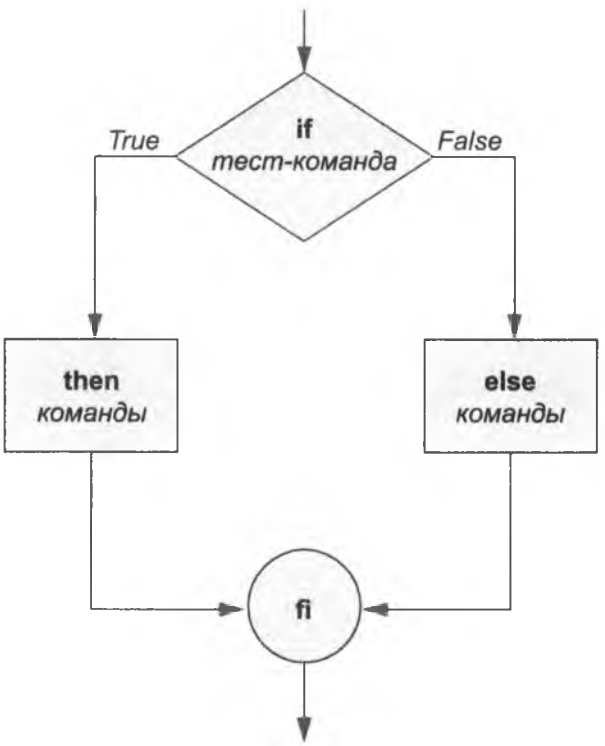

Рис. 10.2. Блок-схема управляющей структуры if...then...else

Благодаря тому что точка с запятой (;) завершает команду точно так же, как это делает разделитель строк, ключевое слово then можно поставить на одну строку с if, поставив перед ним точку с запятой. (Поскольку if и then являются отдельными встроенными командами, между ними нужно ставить разделитель команд; точка с запятой и разделитель строк работают в этом качестве одинаково хорошо [см. раздел «Точка

с запятой (;) и перевод строки (↵) разделяют команды» на с. 296.) Некоторые предпочитают эту систему записи из эстетических соображений, другим она нравится по причине экономии пространства.

```
if тест-команда; then
    команды
else
    команды
fi
```

Если *тест-команда* вернет статус *true*, то структура if выполнит команды, которые находятся между инструкциями then и else, а затем передаст управление инструкции, которая следует за инструкцией fi. Если *тест-команда* вернет статус *false*, структура if выполнит команды, которые следуют за инструкцией else.

При запуске сценарий out с аргументами, представляющими собой имена файлов, выводит содержимое файлов на терминал. Если первым аргументом будет -v (который в данном случае называется ключом), сценарий out использует команду less (см. раздел «less и more: Обеспечение поэкранного вывода текстового файла» на с. 68) для поэкранного вывода содержимого файлов. После определения, что он был вызван как минимум с одним аргументом, сценарий out тестирует свой первый аргумент, чтобы определить, не является ли он ключом -v. Если в результате этого теста будет возвращено значение *true* (первый аргумент является ключом -v), сценарий out использует встроенную команду shift (см. раздел «shift: Смещение аргументов командной строки» на с. 460) для смещения аргументов, чтобы избавиться от -v и вывести содержимое файлов, используя команду less. Если же в результате тестирования будет возвращено значение *false* (первый аргумент не является ключом -v), то сценарий использует для вывода содержимого файлов команду cat:

```
$ cat out
if [ $# -eq 0 ]
    then
        echo "Порядок использования: out [-v] имена_файлов …" 1>&2
        exit 1
fi
if [ "$1" = "-v" ]
    then
        shift
        less -- "$@"
    else
        cat -- "$@"
fi
```

В сценарии out аргумент --, который используется в командах cat и less, сообщает им об отсутствии за ним ключей в командной строке и о том, что лидирующий дефис (-) в последующем списке аргументов не следует рассматривать как признак ключа. Благодаря этому аргумент -- позволяет просматривать содержимое файлов, чьи имена начинаются с дефиса. Такие имена хотя и редко, но попадаются. (Например, такой файл можно создать с помощью команды cat > -имя_файла.) Аргумент -- работает со всеми утилитами Linux, которые используют для разбора своих ключей встроенную команду getopts (см. раздел «getopts: Разбор ключей» на с. 476), но не работает с утилитой more и некоторыми другими утилитами. Этот аргумент особенно полезен, когда используется в связке с командой rm для удаления файла, чье имя начинается с дефиса (rm -- -имя_файла), аргумент -- нужно включать в весь создаваемый вами экспериментальный код.

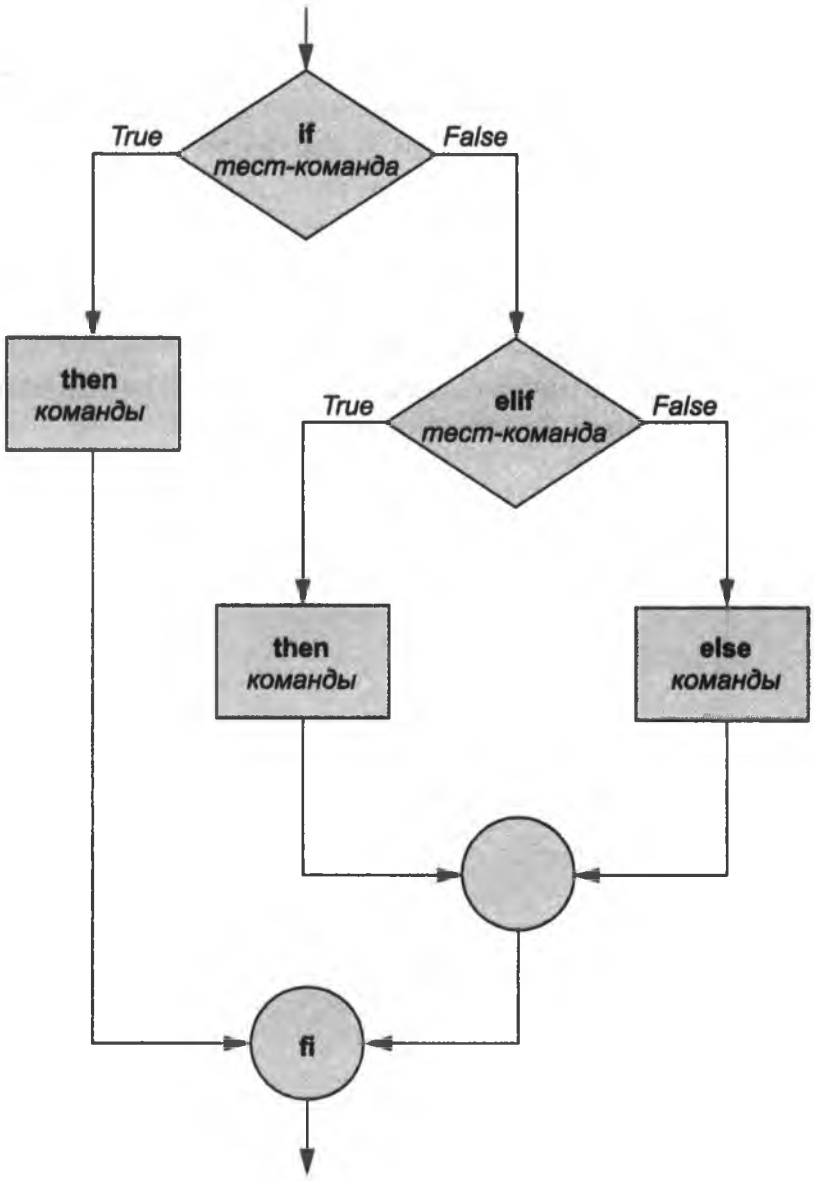

Рис. 10.3. Блок-схема управляющей структуры if...then...elif

Управляющая структура if...then...elif

Для показанной на рис. 10.3 управляющей структуры if...then...elif (в tcsh такая структура отсутствует) используется следующий синтаксис:

```
if тест-команда
   then
      команды
   elif тест-команда
      then
         команды
-
   else
      команды
fi
```

Инструкция elif является сочетанием инструкции else и инструкции if и позволяет конструировать вложенные наборы структур if...then...else (см. рис. 10.3). Разница

между инструкцией else и инструкцией elif состоит в том, что каждая инструкция else должна иметь парную с ней инструкцию fi, в то время как несколько вложенных инструкций elif требуют только одной закрывающей инструкции fi.

Управляющая структура if...then...elif показана в следующем примере. Этот сценарий оболочки сравнивает три слова, вводимые пользователем. Первая инструкция if использует в качестве аргумента для команды test логический оператор И (-a). Встроенная команда test возвращает статус *true* только в том случае, если первое и второе логические сравнения дают результат *true* (то есть word1 соответствует word2, и word2 соответствует word3). Если команда test возвращает статус *true*, сценарий выполняет команды, которые следуют за инструкцией then, передает управление инструкции, которая следует за инструкцией fi, и завершает работу:

```
$ cat if3
echo -n "слово 1: "
read word1
echo -n "слово 2: "
read word2
echo -n "слово 3: "
read word3
if [ "$word1" = "$word2" -a "$word2" = "$word3" ]
    then
        echo "Совпадение: слова 1, 2 & 3"
    elif [ "$word1" = "$word2" ]
    then
        echo "Совпадение: слова 1 & 2"
    elif [ "$word1" = "$word3" ]
    then
        echo "Совпадение: слова 1 & 3"
    elif [ "$word2" = "$word3" ]
    then
        echo "Совпадение: слова 2 & 3"
else
    echo "Совпадений не найдено"
fi
$ ./if3
слово 1: apple
слово 2: orange
слово 3: pear
Совпадений не найдено
$ ./if3
слово 1: apple
слово 2: orange
слово 3: apple
Совпадение: слова 1 & 3
$ ./if3
слово 1: apple
слово 2: apple
слово 3: apple
Совпадение: слова 1, 2 & 3
```

Если эти три слова отличаются друг от друга, структура передает управление первой инструкции elif, которая начинает серию тестов, предназначенных для обнару-

жения совпадений любой пары слов. При продолжении выполнения вложенных инструкций, если любая из этих инструкций if будет удовлетворена, структура передает управление следующей инструкции then, а затем той инструкции, которая следует за инструкцией fi. При каждом случае невыполнения условий инструкции elif структура передает управление следующей инструкции elif. Двойные кавычки, окружающие аргументы команд echo, в которых содержатся знаки амперсанда (&), не позволяют оболочке интерпретировать амперсанд как специальный символ.

СЦЕНАРИЙ LNKS

Следующий сценарий по имени lnks демонстрирует работу управляющих структур if...then и if...then...elif. Этот сценарий ищет жесткие ссылки на свой первый аргумент, в качестве которого выступает имя файла. Если в качестве второго аргумента предоставить имя каталога, lnks ведет поиск ссылок в иерархии каталогов, берущей начало с этого каталога. Если не указать каталог, lnks ведет поиск в рабочем каталоге и его подкаталогах. Символьные ссылки этот сценарий не обнаруживает.

```
$ cat lnks
#!/bin/bash
# Обнаружение ссылок на файл
# Порядок использования: lnks файл [каталог]

if [ $# -eq 0 -o $# -gt 2 ]; then
        echo "Порядок использования: lnks файл [каталог]" 1>&2
        exit 1
fi
if [ -d "$1" ]; then
        echo "Первый аргумент не может быть каталогом." 1>&2
        echo "Порядок использования: lnks файл [каталог]" 1>&2
        exit 1
else
        file="$1"
fi
if [ $# -eq 1 ]; then
            directory="."
        elif [ -d "$2" ]; then
            directory="$2"
        else
            echo "Дополнительным вторым аргументом должен быть каталог." 1>&2
            echo "Порядок использования: lnks файл [каталог]" 1>&2
            exit 1
fi

# Проверка существования файла и его принадлежности к обычным файлам
if [ ! -f "$file" ]; then
        echo "lnks: $file файл не найден или относится к специальным файлам" 1>&2
        exit 1
fi
# Проверка счетчика ссылок на файл
set -- $(ls -l "$file")

linkcnt=$2
if [ "$linkcnt" -eq 1 ]; then
        echo "lnks: других жестких ссылок на файл $file нет" 1>&2
        exit 0
fi
```

```
# Получение i-узла указанного файла
set $(ls -i "$file")

inode=$1

# Поиск файлов с этим номером i-узла и вывод их имен
echo "lnks: использование команды find для поиска ссылок…" 1>&2
find "$directory" -xdev -inum $inode -print
```

У пользователя Max в его личном каталоге есть файл по имени letter. Он хочет найти ссылки на этот файл в своем собственном дереве личного каталога и в деревьях личных каталогов других пользователей. В следующем примере для осуществления поиска Max вызывает сценарий lnks из своего личного каталога. Второй аргумент для сценария lnks, /home, является путевым именем каталога, с которого он хочет начать поиск. При работе с Mac OS X вместо /home нужно подставить /Users. Сценарий lnks сообщает, что файлы /home/max/letter и /home/zach/draft являются ссылками на один и тот же файл:

```
$ ./lnks letter /home
lnks: использование команды find для поиска ссылок…
/home/max/letter
/home/zach/draft
```

Кроме управляющей структуры if...then...elif, в сценарии lnks представлены и другие свойства, часто использующиеся в программах оболочек. Далее сценарий lnks рассматривается по разделам.

Указание оболочки

В первой строке сценария lnks для указания оболочки, которая будет выполнять сценарий, используется последовательность символов #! (см. раздел «#! — способ указания оболочки» на с. 294):

```
#!/bin/bash
```

В этой главе запись #! появляется только в самых сложных примерах. Она обеспечивает выполнение сценария в нужной для него оболочке, даже если пользователь запустил другую оболочку, или сценарий вызван из сценария, запустившего другую оболочку.

Комментарии

Вторая и третья строки сценария lnks являются комментариями, оболочка игнорирует текст, который следует за знаком решетки и продолжается до следующего символа разделителя строк. В lnks эти комментарии кратко описывают, чем занимается файл, и объясняют порядок его использования:

```
# Обнаружение ссылок на файл
    # Порядок использования: lnks файл [каталог]
```

Сообщения о порядке использования

Первая инструкция if тестирует lnks на случай его вызова с нулевым количеством аргументов или с их количеством большим двух:

```
if [ $# -eq 0 -o $# -gt 2 ]; then
        echo "Порядок использования: lnks файл [каталог]" 1>&2
        exit 1
fi
```

Если любое из этих условий вычисляется в *true*, сценарий lnks отправляет на стандартную ошибку сообщение о порядке использования и осуществляет выход со статусом 1. Двойные кавычки, окружающие сообщение о порядке использования, не позволяют оболочке интерпретировать квадратные скобки в качестве специальных символов. Квадратные скобки в сообщении о порядке использования обозначают, что аргумент *каталог* является необязательным.

Вторая инструкция if тестирует, является ли первый аргумент командной строки (то есть $1) каталогом (оператор -d в команде test возвращает *true*, если файл существует и относится к файлу каталога):

```
if [ -d "$1" ]; then
        echo "Первый аргумент не может быть каталогом." 1>&2
```

```
        echo "Порядок использования: lnks файл [каталог]" 1>&2
        exit 1
    else
        file="$1"
fi
```

Если первый аргумент является каталогом, сценарий lnks выводит сообщение о порядке использования и осуществляет выход. Если этот аргумент не является каталогом, то сценарий lnks сохраняет значение переменной `$1` в переменной `file`, потому что позже в этом сценарии команда set устанавливает новые аргументы командной строки. Если значение переменной `$1` не сохранено до выдачи команды set, его значение утрачивается.

Тестирование аргументов

В следующем разделе сценария lnks используется структура if...then...elif:

```
if [ $# -eq 1 ]; then
        directory="."
    elif [ -d "$2" ]; then
        directory="$2"
    else
        echo "Дополнительным вторым аргументом должен быть каталог." 1>&2
        echo "Порядок использования: lnks файл [каталог]" 1>&2
        exit 1
fi
```

Первая *тест-команда* определяет факт указания пользователем в командной строке только одного аргумента. Если *тест-команда* возвращает 0 (*true*), переменной `directory` присваивается значение рабочего каталога (.). Если *тест-команда* возвращает *false*, инструкция elif проверяет отношение второго аргумента к каталогу. Если это каталог, переменной `directory` присваивается значение, равное значению второго аргумента командной строки, `$2`. Если значение `$2` не является именем каталога, сценарий lnks отправляет на стандартную ошибку сообщение о порядке использования и осуществляет выход со статусом 1.

Следующая инструкция if в сценарии lnks проверяет наличие файла с именем, хранящемся в переменной `$file`. Этот тест избавляет сценарий lnks от напрасной траты времени для поиска ссылок на несуществующий файл. Встроенная команда test, вызываемая с тремя аргументами: `!`, `-f` и `$file`, вычисляется в *true*, если файл, чье имя хранится в переменной `$file`, не существует:

```
[ ! -f "$file" ]
```

Оператор `!`, находящийся перед аргументом `-f` команды test, инвертирует результат ее работы, выдавая *false*, если файл с именем, хранящемся в переменной `$file`, существует и является обычным файлом.

Затем для проверки количества ссылок, имеющихся у файла, имя которого хранится в переменной `$file`, в сценарии lnks используется команда set и команда `ls -l`:

```
# Проверка счетчика ссылок на файл
set -- $(ls -l "$file")

linkcnt=$2
if [ "$linkcnt" -eq 1 ]; then
        echo "lnks: других жестких ссылок на файл $file нет" 1>&2
        exit 0
fi
```

Встроенная команда set использует командную подстановку (см. раздел «Командная подстановка» на с. 358) для установки позиционного параметра на вывод команды `ls -l`. Второе поле в этом выводе является счетчиком ссылок, поэтому пользовательской переменной linkcnt присваивается значение, равное значению переменной `$2`. Аргумент `--`, используемый с командой set, не дает ей интерпретировать первый аргумент, создающийся командой `ls -l`, как ключ (первым аргументом служит индикатор прав доступа к файлу, который обычно начинается с символа `-`).

Инструкция if проверяет, не равно ли значение переменной $linkcnt единице, при этом равенстве сценарий lnks выводит сообщение и осуществляет выход. Хотя это сообщение и не является настоящим сообщением об ошибке, сценарий перенаправляет его на стандартную ошибку. Сценарий lnks написан так, что все информационные сообщения перенаправляются на стандартную ошибку. Финальный продукт сценария lnks — путевые имена ссылок на указанный файл — отправляется на стандартный вывод, поэтому вывод можно перенаправить куда угодно.

Если счетчик ссылок содержит значение, большее 1, сценарий lnks переходит к идентификации i-узла для файла, имя которого хранится в переменной $file. Согласно объяснениям, приведенным в разделе «Утилита ls и i-узлы» на с. 129, сравнение i-узлов, связанных с именами файлов, является неплохим способом определения, являются ли имена файлов ссылками на один и тот же файл. (При использовании файловой системы HFS+ исходные структуры данных отличаются, но система ведет себя аналогичным образом.) Сценарий lnks использует команду set для установки позиционных параметров на значение вывода команды ls -i. Первым аргументом для команды set служит номер i-узла для файла, поэтому пользовательской переменной по имени inode присваивается значение переменной $1:

```
# Получение i-узла указанного файла
set $(ls -i "$file")
inode=$1
```

В самом конце сценарий lnks использует для поиска файлов, имеющих номера i-узлов, соответствующих значению переменной $inode, утилиту find (см. часть V):

```
# Поиск файлов с этим номером i-узла и вывод их имен
echo "lnks: использование команды find для поиска ссылок..." 1>&2
find "$directory" -xdev -inum $inode -print
```

Утилита find ведет поиск тех файлов, которые отвечают критериям, указанным в остальных аргументах, в иерархии каталогов, которая берет начало в каталоге, указанном в качестве ее первого аргумента ($directory). В данном примере оставшиеся аргументы отправляют имена файлов, имеющих i-узлы, соответствующие значению переменной $inode, на стандартный вывод. Поскольку файлы в различных файловых системах могут иметь одинаковые номера i-узлов и при этом могут быть не связаны друг с другом, утилита find должна вести поиск только в тех каталогах, которые относятся к той же файловой системе, что и каталог, указанный в переменной $directory. Аргумент -xdev (исключение поиска на всех устройствах) препятствует тому, чтобы утилита find вела поиск в каталогах на других файловых системах. Дополнительные сведения о файловых системах и ссылках даны в разделе «Ссылки» на с. 126.

Команда echo, предшествующая в сценарии lnks команде find и сообщающая пользователю, что работает команда find, включена в сценарий потому, что find может выполняться достаточно долго. Поскольку сценарий lnks не включает в себя завершающую инструкцию exit, статус выхода у lnks такой же, как и у последней запускаемой им команды find.

Отладка сценариев оболочки

При написании таких сценариев, как lnks, можно легко ошибиться. Чтобы облегчить отладку сценария, можно воспользоваться ключом оболочки –x. Этот ключ заставляет оболочку выводить каждую команду перед ее запуском. Такая трассировка выполнения сценария может предоставить вам информацию об источнике проблемы.

Можно запустить сценарий lnks так же, как и в предыдущем примере, и заставить оболочку выводить каждую команду перед ее выполнением. Для этого нужно либо установить ключ –x для текущей оболочки (set -x), и тогда все сценарии будут выводить команды по мере их запуска, либо воспользоваться ключом -x для воздействия только на ту оболочку, которая выполняет данный сценарий.

```
$ bash -x lnks letter /home
+ '[' 2 -eq 0 -o 2 -gt 2 ']'
+ '[' -d letter ']'
+ file=letter
+ '[' 2 -eq 1 ']'
+ '[' -d /home ']'
+ directory=/home
+ '[' '!' -f letter ']'
...
```

PS4 Каждая команда, выполняемая сценарием, предваряется значением, хранящимся в переменной PS4. По умолчанию это знак плюс (+). Это дает возможность отличить отладочный вывод от вывода, осуществляемого сценарием. Для установки значения переменной PS4 в оболочке, вызывающей сценарий, эту переменную нужно экспортировать. Следующая команда устанавливает для переменной PS4 значение >>>>, за которым следует пробел, и экспортирует эту переменную:

```
$ export PS4='>>>> '
```

Можно также установить ключ -x для оболочки, выполняющий сценарий, разместив где-нибудь в начале сценария следующую команду set:

```
set -x
```

Если нужно включить отладку, следует поставить команду set -x в соответствующем месте сценария. Выключение отладки производится с помощью знака плюс:

```
set +x
```

Команды set -o xtrace и set +o xtrace делают то же самое, что и команды set -x и set +x соответственно.

Управляющая структура for...in

Для управляющей структуры for...in (в tcsh используется foreach) применяется следующий синтаксис:

```
for индекс_цикла in список_аргументов
do
    команды
done
```

Структура for...in (рис. 10.4) присваивает значение первого аргумента из *списка_аргументов* переменной *индекс_цикла* и выполняет *команды*, которые находятся между инструкциями do и done. Инструкции do и done обозначают начало и конец цикла.

После того как *команды* передают управление инструкции done, структура присваивает *индексу_цикла* значение второго аргумента из *списка_аргументов* и повторяет выполнение *команд*. Затем она повторяет выполнение *команд* между инструкциями do и done по одному разу для каждого аргумента, имеющегося в *списке_аргументов*. Когда структура исчерпает *список_аргументов*, она передает управление инструкции, которая следует за инструкцией done.

Следующая структура for...in присваивает значение apples пользовательской переменной fruit, а затем выводит значение переменной fruit, которое равно apples. Затем структура присваивает переменной fruit значение oranges и повторяет процесс. Когда

Рис. 10.4. Блок-схема управляющей структуры for...in

список аргументов будет исчерпан, структура передает управление инструкции, которая следует за инструкцией done, а эта инструкция выводит сообщение:

```
$ cat fruit
for fruit in apples oranges pears bananas
do
    echo "$fruit"
done
echo "Задача завершена."

$ ./fruit
apples
oranges
pears
bananas
Задача завершена.
```

Следующий сценарий выводит список имен каталогов, имеющихся в рабочем каталоге, осуществляя циклический перебор имен файлов, находящихся в рабочем каталоге, и используя команду test для определения, какие из них являются каталогами:

```
$ cat dirfiles
for i in *
do
    if [ -d "$i" ]
        then
            echo "$i"
fi
done
```

Нестрогая ссылка на имена файлов * соответствует именам всех файлов (за исключением скрытых) в рабочем каталоге. Перед выполнением цикла оболочка расширяет ссылку * и использует получившийся список для присваивания последовательных значений индексной переменной `i`.

Управляющая структура for

Для управляющей структуры for (отсутствующей в tcsh) используется следующий синтаксис:

```
for индекс_цикла
do
    команды
done
```

В структуре for *индекс_цикла* поочередно принимает значение каждого аргумента командной строки. Структура for такая же структура, как и for...in (см. рис 10.4), за исключением определения источника получения значений для *индекса_цикла*. Структура for выполняет последовательность *команд*, обычно привлекая по очереди значение каждого аргумента.

Следующий сценарий оболочки показывает структуру for, которая выводит каждый аргумент командной строки. В первой строке сценария, `for arg`, подразумевается `for arg in "$@"`, где оболочка расширяет переменную `"$@"` в список взятых в кавычки аргументов командной строки `"$1" "$2" "$3"` и т. д. В остальном сценарий соответствует структуре for...in.

```
$ cat for_test
for arg
do
   echo "$arg"
done

$ for_test candy gum chocolate
candy
gum
chocolate
```

СЦЕНАРИЙ WHOS

Следующий сценарий по имени whos показывает, как можно извлечь пользу из подразумеваемого в структуре if элемента `"$@"`. Сценарию whos в качестве аргументов предоставляется одно или несколько полных имен пользователей или пользовательских имен, и whos выводит информацию о пользователях. Сценарий whos получает выводимую им информацию из первого и пятого полей, имеющихся в файле /etc/passwd. В первом поле содержится имя пользователя, а в пятом поле обычно содержится полное имя пользователя.

В качестве аргумента сценарию whos можно предоставить имя пользователя, чтобы он идентифицировал настоящее имя этого пользователя, или же ему можно предоставить в качестве аргумента настоящее имя, чтобы он идентифицировал имя пользователя. Сценарий whos похож на утилиту finger, хотя whos предоставляет меньший объем информации. В Mac OS X вместо файла passwd используется Open Directory.

```
$ cat whos
#!/bin/bash

if [ $# -eq 0 ]
       then
           echo "Порядок использования: whos идентификатор…" 1>&2
           exit 1
fi
for id
do
       gawk -F: '{print $1, $5}' /etc/passwd |
       grep -i "$id"
done
```

В следующем примере сценарий whos идентифицирует пользователя, чье пользовательское имя chas, и пользователя, чье настоящее имя Marilou Smith:

```
$ ./whos chas "Marilou Smith"
chas Charles Casey
msmith Marilou Smith
```

Использование переменной «$@» В сценарии whos инструкция for используется для последовательного перебора аргументов командной строки. В этом сценарии в цикле for подразумеваемое использование переменной `"$@"` особенно выгодно, поскольку оно заставляет цикл for рассматривать аргументы, содержащие пробел в качестве единого аргумента.

В данном примере имя `Marilou Smith` взято в кавычки, что заставляет оболочку передавать его сценарию как единый аргумент. Затем подразумеваемая для цикла **for** переменная `"$@"` заставляет оболочку регенерировать заключенный в кавычки аргумент `Marilou Smith`, благодаря чему он опять рассматривается в качестве единого аргумента.

gawk

Для каждого аргумента командной строки сценарий **whos** осуществляет поиск в файле **/etc/passwd**. Внутри цикла **for** утилита **gawk** (в главе 12 точно так же будут работать утилиты **awk** и **mawk**) извлекает первые (`$1`) и пятые (`$5`) поля из каждой строки файла **/etc/passwd**. Ключ `-F:` заставляет утилиту **mawk** использовать в качестве разделителя полей при чтении файла **/etc/passwd** двоеточие (`:`), позволяя ей разбивать каждую строку на поля. Команда **mawk** устанавливает и использует значение аргументов `$1` и `$5`; они заключены в одинарные кавычки и не интерпретируются оболочкой. Эти аргументы не следует путать с позиционными параметрами, которые соответствуют аргументам командной строки. Первое и пятое поля отправляются через канал команде **grep** (см. часть V). Утилита **grep** ищет в своем вводе значение переменной `$id` (которой оболочка присвоила значение аргумента командной строки). Ключ `-i` заставляет **grep** игнорировать в процессе поиска регистр символов. Утилита **grep** выводит каждую строку из своего ввода, в которой содержится значение переменной `$id`.

Символ | в конце строки

В строке, содержащей команду **mawk**, используется интересное синтаксическое исключение, которое оболочка **bash** делает для символа канала (`|`): вам не нужно цитировать (или отключать) символ разделителя строк, который следует сразу же за символом канала (то есть за символом канала, который является последним символом в строке), чтобы подавить свойство разделителя строк, запускающее команду на выполнение. Попробуйте набрать команду `who |` и нажать **ВВОД**. Оболочка (только не **tcsh**) выведет вторичное приглашение. Если затем ввести команду **sort** и снова нажать **ВВОД**, то вы увидите отсортированный список, выведенный командой **who**. Канал работает даже в том случае, если за его символом следует разделитель строк.

Управляющая структура while

Для управляющей структуры **while** (которая недоступна в **tcsh**) используется следующий синтаксис:

```
while тест-команда
    do
        команды
    done
```

Пока *тест-команда* (рис. 10.5) возвращает статус выхода *true*, структура **while** продолжает выполнять ряд команд, ограниченный инструкциями **do** и **done**. Перед каждым проходом команд структура выполняет *тест-команду*. Если статус выхода *тест-команды* будет равен *false*, структура передает управление той инструкции, которая следует за инструкцией **done**.

Встроенная команда test

Следующий сценарий оболочки сначала инициализирует переменную `number`, присваивая ей нулевое значение. Затем встроенная команда **test** определяет, не является ли значение переменной `number` меньше 10. Чтобы провести числовое тестирование, сценарий использует команду **test** с аргументом `-lt`. Для сравнения чисел нужно использовать аргументы `-ne` (not equal — не равно), `-eq` (equal – равно), `-gt` (greater than — больше чем), `-ge` (greater than or equal to — больше чем или

равно), -lt (less than — меньше чем) или -le (less than or equal to — меньше чем или равно). Для сравнения строк при работе с командой test нужно использовать аргументы = (равно) или != (не равно). В данном примере у команды test статус выхода равен 0 (*true*) до тех пор, пока значение переменной number меньше 10. Пока команда test возвращает *true*, структура выполняет команды, которые находятся между инструкциями do и done. В части V дается описание утилиты test, которая очень похожа на встроенную команду test.

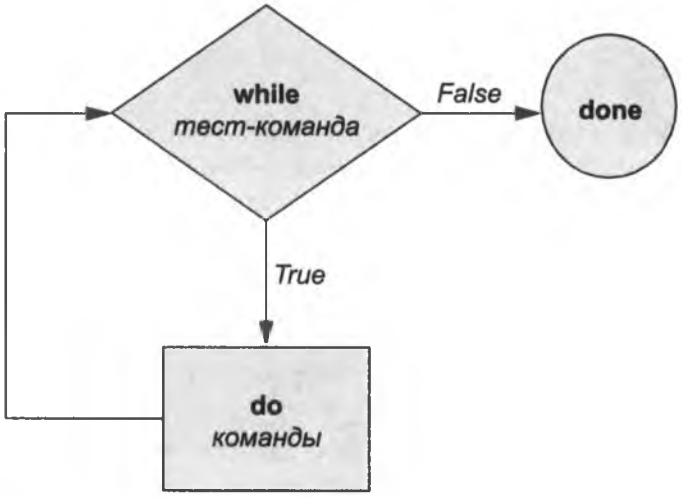

Рис. 10.5. Блок-схема управляющей структуры while

```
$ cat count
#!/bin/bash
number=0
while [ "$number" -lt 10 ]
   do
      echo -n "$number"
      ((number +=1))
   done
echo
$ ./count
0123456789
$
```

Команда echo, которая следует за инструкцией do, выводит значение переменной number. Ключ -n препятствует выводу командой echo разделителя строк после вывода данных. В следующей команде используется арифметическое вычисление (((…)); см. раздел «Арифметические вычисления» на с. 480) для увеличения на единицу значения переменной number. Инструкция done завершает цикл и возвращает управление инструкции while для очередного начала цикла. Завершающая команда echo заставляет сценарий count отправить на стандартный вывод разделитель строк, поэтому следующее приглашение появляется на дисплее в самой крайней левой позиции (а не сразу же за цифрой 9).

СЦЕНАРИЙ SPELL_CHECK

Утилита aspell проверяет слова в файле по словарю на правильность их написания. При вызове с аргументом `list` команда aspell (только в Linux) запускается в режиме списка: ввод поступает со стандартного ввода, и aspell отправляет каждое потенциально неправильно написанное слово на стандартный вывод.

Следующая команда выдает список возможных неправильно написанных слов, найденных в файле letter.txt:

```
$ aspell list < letter.txt
quikly
portible
frendly
```

Следующий сценарий оболочки по имени spell_check демонстрирует еще один способ использования структуры while. Чтобы найти в файле неправильно написанные слова, сценарий spell_check вызывает утилиту aspell для проверки содержимого файла по словарю системы. Но этот сценарий имеет и более широкие возможности: позволяет указать список правильно написанных слов и удалить эти слова из вывода утилиты aspell. Этот сценарий может пригодиться для удаления часто используемых слов, таких как имена и технические термины, отсутствующие в стандартном словаре. Конечно, функциональные возможности сценария spell_check можно реализовать путем использования для утилиты aspell дополнительных словарей, поэтому данный сценарий включен в книгу в чисто учебных целях.

Сценарий spell_check требует в качестве аргументов предоставить ему два имени файла: имя того файла, в котором содержится список правильно написанных слов, и имя того файла, содержимое которого нужно проверить. Первая инструкция if осуществляет проверку факта предоставления пользователем двух аргументов. Следующие две инструкции if проверяют отношение обоих аргументов к именам читаемых файлов. (Восклицательный знак инвертирует смысл следующих операторов; оператор `-r` заставляет команду test определить, является ли файл читаемым. В результате получается тест, который определяет, является ли файл *нечитаемым*.)

```
$ cat spell_check
#!/bin/bash
# удаление правильно написанных слов из вывода утилиты aspell

if [ $# -ne 2 ]
        then
            echo "Порядок использования: spell_check файл1 файл2" 1>&2
         echo "файл1: список правильно написанных слов" 1>&2
              echo "файл2: проверяемый файл " 1>&2
            exit 1
fi

if [ ! -r "$1" ]
        then
        echo "spell_check: $1 не читается" 1>&2
        exit 1
fi

if [ ! -r "$2" ]
        then
        echo "spell_check: $2 не читается" 1>&2
        exit 1
fi

aspell list < "$2" |
while read line
    do
```

```
    if ! grep "^$line$" "$1" > /dev/null
        then
            echo $line
    fi
done
```

Сценарий spell_check отправляет вывод утилиты aspell (которая используется с аргументом list и поэтому выдает на стандартный вывод список неправильно написанных слов) через канал на стандартный ввод структуры while, которая производит построчное чтение (в каждой строке содержится одно слово) из стандартного ввода. *Тест-команда* (то есть read line) возвращает статус выхода *true*, пока она получает строку из стандартного ввода.

Внутри цикла while возвращаемое значение команды grep отслеживается инструкцией if[1], которая определяет, не фигурирует ли прочитанная строка в списке правильно написанных слов. Задающий поиск шаблон команды grep (значение переменной $line) содержит в начале и в конце специальные символы, которые определяют начало и конец строки (соответственно ^ и $). Эти специальные символы гарантируют, что команда grep ищет соответствие только в том случае, если содержимое переменной $line соответствует целиком всей строке файла правильно написанных слов. (В противном случае команда grep будет находить соответствие в выводе утилиты aspell такому слову, как paul, если в файле правильно написанных слов будет присутствовать слово paulson.) Эти специальные символы вместе со значением переменной $line составляют регулярное выражение.

Вывод команды grep перенаправляется за ненадобностью в файл /dev/null (см. раздел «/dev/null: отправка данных в никуда» на с. 151), здесь важен только код выхода. Инструкция if осуществляет проверку инвертированного статуса выхода из команды grep (лидирующий восклицательный знак инвертирует или изменяет смысл статуса выхода, и *true* становится *false*, и наоборот), который равен 0 или *true* (а после инвертирования он равен *false*), когда находится соответствующая строка. Если статус выхода не равен 0 или равен *false* (*true* после инвертирования), значит, это слово не присутствует в файле правильно написанных слов. Встроенная команда echo отправляет список слов, которых нет в файле правильно написанных слов, в стандартный вывод.

Как только будет обнаружен символ EOF (end of file — конец файла), встроенная команда read вернет статус выхода *false*, управление будет передано за пределы структуры while, и сценарий завершит работу. Перед использованием сценария spell_check нужно создать файл правильно написанных слов, содержащий часто используемые, но отсутствующие в стандартном словаре слова. Например, если вы работаете на компанию, которая называется «Blinkenship and Klimowski, Attorneys», в этот файл нужно поместить слова Blinkenship и Klimowski. В следующем примере показано, как сценарий spell_check проверяет правописание слов в файле по имени memo и удаляет Blinkenship и Klimowski из выводимого списка неправильно написанных слов:

```
$ aspell list < memo
Blinkenship
Klimowski
targat
hte
$ cat word_list
Blinkenship
Klimowski
$ ./spell_check word_list memo
targat
hte
```

Дополнительные сведения об утилите aspell (которая имеется только в Linux) даны в части V.

---

[1] Эта инструкция if может также быть написана следующим образом:

```
if ! grep -qw "$line" "$1"
```

Ключ -q подавляет вывод из команды grep, поэтому она возвращает только код выхода. Ключ -w заставляет команду grep искать соответствие только в целом слове.

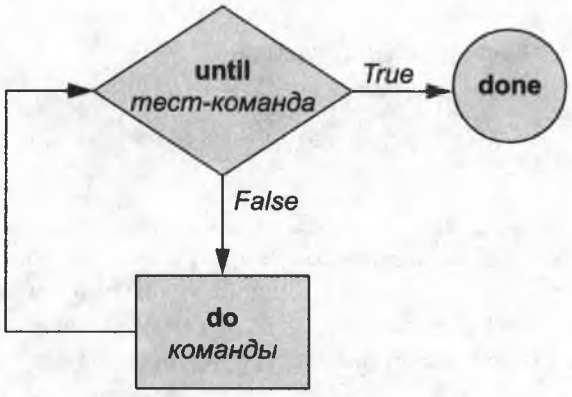

Рис. 10.6. Блок-схема управляющей структуры until

Управляющая структура until

Структуры until (отсутствующая в tcsh) и while (присутствующая в tcsh с немного другим синтаксисом) очень похожи друг на друга и различаются только в смысле проводимого теста в самом начале цикла. На рис. 10.6 показано, что структура until продолжает выполнение цикла до тех пор, *пока тест-команда* не вернет статус выхода *true*. А структура while выполняет цикл до тех пор, *пока тест-команда* продолжает возвращать *true* или не возникает ошибка. Для управляющей структуры until используется следующий синтаксис:

until ***тест-команда***
do
    команды
done

Следующий сценарий показывает структуру until, которая включает в себя команду read. Когда пользователь вводит правильную строку символов, *тест-команда* выдает положительный результат, и структура передает управление за пределы цикла.

```
$ cat until1
secretname=zach
name=noname
echo "Попробуйте отгадать секретное имя!"
echo
until [ "$name" = "$secretname" ]
do
    echo -n "Ваша отгадка: "
    read name
done
echo "Очень хорошо."

$ ./until1
Попробуйте отгадать секретное имя!

Ваша отгадка: helen
Ваша отгадка: barbara
Ваша отгадка: rachael
Ваша отгадка: zach
Очень хорошо.
```

Следующий сценарий locktty похож на команду lock операционной системы Berkeley UNIX и на пункт меню Lock Screen (Блокировка экрана) в интерфейсе GNOME. Сценарий запрашивает ключ (пароль) и использует управляющую структуру until для блокировки терминала. Инструкция until заставляет систему игнорировать любые введенные с клавиатуры символы до тех пор, пока пользователь самостоятельно не наберет в строке ключ, разблокирующий терминал, за которым последует нажатие клавиши ВВОД. Сценарий locktty может не допустить использования терминала на время вашего кратковременного отсутствия. Он избавляет вас от необходимости выходить из системы, если вы обеспокоены тем, что кто-то может воспользоваться вашим сеансом работы.

```
$ cat locktty
#! /bin/bash

trap '' 1 2 3 18
stty -echo
echo -n "Ключ: "
read key_1
echo
echo -n "Еще раз: "
read key_2
echo
key_3=
if [ "$key_1" = "$key_2" ]
   then
      tput clear
      until [ "$key_3" = "$key_2" ]
      do
         read key_3
      done
   else
      echo "locktty: ключи не совпадают" 1>&2
fi
stty echo
```

ВЫ ЗАБЫЛИ СВОЙ ПАРОЛЬ ДЛЯ LOCKTTY?

Если вы забудете свой ключ (пароль), нужно будет зайти в систему с другого (виртуального) терминала и прекратить процесс, в котором запущен сценарий locktty.

Встроенная команда trap

Встроенная команда trap, отсутствующая в оболочке tcsh (см. раздел «trap: Перехват сигнала» на с. 472), которая находится в самом начале сценария locktty, исключает для пользователя возможность прекращения выполнения сценария путем отправки ему сигнала (например, путем нажатия клавиши прерывания). Перехват сигнала 18 означает, что никто не сможет воспользоваться комбинацией CONTROL+Z (управляющей выполнением задания, посылающей команду остановки с терминала tty), чтобы сломать блокировку. Список сигналов приведен в табл. 10.5. Команда stty -echo (см. часть V) заставляет терминал не выводить символы, набираемые на клавиатуре, препятствуя появлению на экране ключа, вводимого пользователем. После выключения вывода набираемых на клавиатуре символов сценарий запрашивает у пользователя ключ, считывает его в пользовательскую пере-

менную key_1, запрашивает у пользователя повторный ввод того же самого ключа и сохраняет его в переменной key_2. Инструкция key_3= создает переменную со значением NULL. Если значения переменных key_1 и key_2 совпадают, сценарий locktty очищает экран (с помощью команды tput) и запускает цикл until. Этот цикл продолжает попытки считывания данных с терминала и присваивания ввода переменной key_3. Как только пользователь наберет строку, которая соответствует значению одного из исходных ключей (key_2), цикл until завершится, и будет опять включен вывод набираемых на клавиатуре символов.

Инструкции break и continue

Выполнение циклов for, while или until можно прекратить с помощью инструкции break или инструкции continue. Инструкция break передает управление той инструкции, которая следует за инструкцией done, прекращая таким образом выполнение цикла. А инструкция continue передает управление инструкции done, продолжая тем самым выполнение цикла.

Использование этих двух инструкций показано в следующем сценарии. Структура for...in осуществляет циклический перебор значений от 1 до 10. Первая инструкция if выполняет свои команды, когда значение переменной index меньше или равно 3 ($index -le 3). Вторая инструкция if выполняет свои команды, когда значение переменной index больше или равно 8 ($index -ge 8). Между этими двумя структурами if команда echo выводит значение переменной index. Для всех значений до 3 включительно первая инструкция if выводит слово continue, выполняет инструкцию continue, которая пропускает команду echo $index и вторую инструкцию if, и продолжает работу с очередной инструкцией for. Для значения, равного 8, вторая инструкция if выводит слово break и выполняет инструкцию break, которая осуществляет выход из цикла.

```
$ cat brk
for index in 1 2 3 4 5 6 7 8 9 10
    do
        if [ $index -le 3 ] ; then
            echo "continue"
            continue
        fi
#
    echo $index
#
    if [ $index -ge 8 ] ; then
        echo "break"
        break
    fi
done

$ ./brk
continue
continue
continue
4
5
```

```
6
7
8
break
```

Управляющая структура case

Управляющая структура case (см. рис. 10.7) является механизмом принятия решений с многократным разветвлением. Путь, который пролегает через структуру, зависит от соответствия или отсутствия соответствия между *тест-строкой* и одним из *шаблонов*. Для управляющей структуры case (в tcsh используется switch) применяется следующий синтаксис:

```
case тест-строка in
   шаблон-1)
      команды-1
      ;;
   шаблон-2)
      команды-2
      ;;
   шаблон-3)
      команды-3
      ;;
...
esac
```

Следующая структура case проверяет введенный пользователем символ в качестве *тест-строки*. Это значение сохраняется в переменной letter. Если *тест-строка* имеет значение A, структура выполняет *команды*, которые следуют за *шаблоном* A. Правые круглые скобки являются частью синтаксиса управляющей структуры case, а не частью *шаблона*. Если *тест-строка* имеет значение B или C, структура выполняет команды, которые следуют за соответствующим шаблоном. Звездочка (*) означает *любая строка символов* и служит универсальной ловушкой в случае, если не найдено никаких соответствий. Если *тест-строке* не соответствует ни один *шаблон* и если нет шаблона универсальной ловушки (*), управление передается команде, которая следует за инструкцией esac, без каких-либо действий со стороны структуры case.

```
$ cat case1
echo -n "Введите A, B или C: "
read letter
case "$letter" in
   A)
      echo "Вы ввели A"
      ;;
   B)
      echo " Вы ввели B"
      ;;
   C)
      echo " Вы ввели C"
      ;;
   *)
```

```
        echo "Вы не ввели ни A, ни B, ни C"
        ;;
esac

$ ./case1
Введите A, B или C: B
Вы ввели B
```

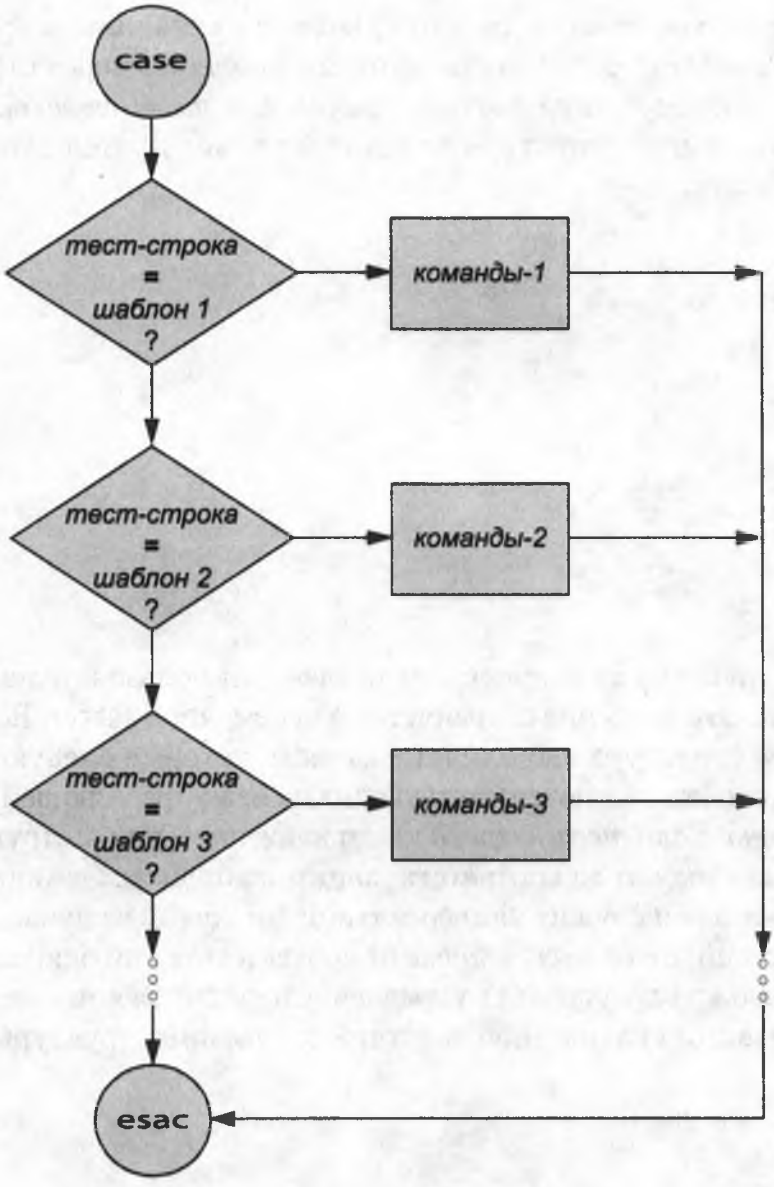

Рис. 10.7. Блок-схема управляющей структуры case

В следующем выполнении сценария case1 показано, что пользователь ввел букву b в нижнем регистре. Поскольку *тест-строка* b не соответствует *шаблону*, состоящему из буквы B верхнего регистра (или любому другому *шаблону* в структуре case), программа выполняет *команды*, которые следуют за *шаблоном* универсальной ловушки, и выводит сообщение:

```
$ ./case1
Введите A, B или C: b
Вы не ввели ни A, ни B, ни C
```

Шаблон в структуре case аналогичен нестрогой ссылке на имена файлов. Он может включать в себя любые специальные символы и строки, показанные в табл. 10.2.

Следующий сценарий принимает буквы как в верхнем, так и в нижнем регистре:

```
$ cat case2
echo -n "Введите A, B или C: "
read letter
case "$letter" in
   a|A)
      echo "Вы ввели A"
      ;;
   b|B)
      echo "Вы ввели B"
      ;;
   c|C)
      echo "Вы ввели C"
      ;;
   *)
      echo "Вы не ввели ни A, ни B, ни C "
      ;;
esac

$ ./case2
Введите A, B или C: b
Вы ввели B
```

Таблица 10.2. Шаблоны

Шаблон	Функция
*	Соответствует любой строке символов. Используется для случая несоответствия ни одному из предыдущих шаблонов
?	Соответствует любому одному символу
[…]	Определяет класс символов. Поочередно проверяется соответствие любому одному символу, заключенному в квадратные скобки. Дефис между двумя символами задает диапазон символов
|	Разделяет альтернативные варианты, которые удовлетворяют конкретной ветви структуры case

В следующем примере показано, как можно использовать структуру case для создания простого меню. Сценарий command_menu использует команду echo для предоставления пунктов меню и приглашения пользователю сделать выбор. (Создать меню намного проще с помощью рассматриваемой далее управляющей структуры select.) Затем структура case в зависимости от сделанного пользователем выбора запускает соответствующую утилиту.

```
$ cat command_menu
#!/bin/bash
# меню для запуска простых команд

echo -e "\n      МЕНЮ КОМАНД \n"
echo "  a.  Тещущая дата и время"
echo "  b.  Пользователи, находящиеся в данный момент в системе"
echo "  c.  Имя рабочего каталога"
echo -e "  d.  Содержимое рабочего каталога\n"
```

```
echo -n "Введите a, b, c или d: "
read answer
echo
#
case "$answer" in
    a)
        date
        ;;
    b)
        who
        ;;
    c)
        pwd
        ;;
    d)
        ls
        ;;
    *)
        echo "Неверный выбор: $answer"
        ;;
esac

$ ./command_menu

МЕНЮ КОМАНД
a.  Текущая дата и время
b.  Пользователи, находящиеся в данный момент в системе
c.  Имя рабочего каталога
d.  Содержимое рабочего каталога
Введите a, b, c или d: a

Wed Jan  3 12:31:12 PST 2009
```

echo -e Ключ -e заставляет echo интерпретировать символы \n как разделитель строк. Если не использовать этот ключ, команда echo не выведет дополнительные пустые строки, а вместо них выведет (буквально) двухсимвольную последовательность \n. Ключ -e заставляет команду echo интерпретировать также и некоторые другие, процитированные с помощью обратного слэша символы (табл. 10.3). Чтобы оболочка их не интерпретировала, а просто передавала обратный слэш команде echo, не забудьте взять их в кавычки (то есть поставить вокруг строки двойные кавычки). Свойство xpg_echo (см. табл 8.13) позволяет обойтись без использования ключа -e.

Таблица 10.3. Специальные символы в echo (следует использовать ключ -e)

Процитированный символ	Вывод команды echo
\a	Сигнал тревоги (звонок)
\b	BACKSPACE
\c	Подавление замыкающего разделителя строк
\f	Перевод страницы (FORMFEED)
\n	Разделитель строк (NEWLINE)
\r	ВВОД (RETURN)
\t	Горизонтальная табуляция

Процитированный символ	Вывод команды echo
\v	Вертикальная табуляция
\\	Обратный слэш
\nnn	Символ с восьмеричным кодом ASCII *nnn*, если *nnn* не является настоящим кодом, команда echo выводит строку в буквальном виде

Управляющую структуру case можно также использовать для задания сценарию различных действий в зависимости от того, с каким количеством аргументов он был вызван. В следующем сценарии safedit используется структура case, которая осуществляет разветвление в зависимости от количества аргументов командной строки ($#). Этот сценарий сохраняет резервную копию файла, редактируемого в vim.

```
$ cat safedit
#!/bin/bash

PATH=/bin:/usr/bin
script=$(basename $0)

case $# in
       0)
           vim
           exit 0
           ;;
       1)
           if [ ! -f "$1" ]
               then
                   vim "$1"
                   exit 0
               fi
           if [ ! -r "$1" -o ! -w "$1" ]
               then
                   echo "$script: Проверка прав доступа к $1" 1>&2
                   exit 1
               else
                   editfile=$1
               fi
           if [ ! -w "." ]
               then
                   echo "$script: Создание резервной копии " \
                       "в рабочем каталоге невозможно" 1>&2
                   exit 1
               fi
               ;;
       *)
               echo "Порядок использования: $script [редактируемый_файл]" 1>&2
               exit 1
               ;;
esac
tempfile=/tmp/$$.$script
cp $editfile $tempfile
if vim $editfile
   then
       mv $tempfile bak.$(basename $editfile)
```

```
        echo "$script: Резервная копия файла создана"
    else
        mv $tempfile editerr
        echo "$script: ошибка редактирования -- копия " \
            "исходного файла находится в editerr" 1>&2
fi
```

Если вызвать safedit без аргументов, case выполнит свою первую ветвь и вызовет редактор vim без файла-аргумента. Поскольку редактирования существующего файла не произошло, сценарий safedit не создает резервную копию файла. (Способ выхода из vim, когда он вызван без имени файла, рассмотрен в описании команды :w в разделе «Аварийное завершение сеанса редактирования» на с. 182.)

Когда сценарий safedit вызван с одним аргументом, он запускает команду из второй ветви структуры case и осуществляет проверку существования файла, чье имя задано в переменной $1, или наличия у пользователя прав на чтение и запись этого файла. Сценарий safedit также проверяет, что у пользователя есть право на запись в рабочий каталог. Если пользователь вызвал safedit с более чем одним аргументом, третья ветвь структуры case выводит сообщение о порядке использования и осуществляет выход со статусом 1.

Установка значения переменной PATH

В дополнение к использованию структуры case для разветвления на основе количества аргументов командной строки, сценарий safedit содержит ряд других особенностей. В начале сценария значение переменной PATH настраивается на задание поиска в каталогах /bin и /usr/bin. Подобная установка значения PATH гарантирует выполнение сценарием тех команд, которые являются стандартными утилитами, присутствующими в этих каталогах. Установка значения этой переменной из сценария позволяет избежать возможных проблем, возникших из-за того, что пользователь установил значение PATH на поиск в первую очередь по собственным каталогам. Для достижения такого же результата можно включить в сценарий абсолютные имена, хотя такая практика может снизить переносимость сценария на другие машины.

Имя программы

В следующей строке создается переменная по имени script и используется командная подстановка для присваивания этой переменной простого имени того файла, в котором содержится сценарий:

```
script=$(basename $0)
```

Утилита basename отправляет на стандартный вывод компонент ее аргумента, представляющий собой простое имя файла, который присваивается переменной script с помощью командной подстановки. Переменная $0 содержит команду, с помощью которой был вызван сценарий (см. раздел «$0: Имя вызванной программы» на с. 459). Независимо от того, какой из следующих команд пользователь вызвал сценарий, выводом команды basename будет простое имя файла safedit:

```
$ /home/max/bin/safedit memo
$ ./safedit memo
$ safedit memo
```

После установки значения переменной script им заменяется имя файла сценария в сообщениях об ошибках и о порядке использования. Благодаря использованию переменной, получающей свое значение из команды, вызвавшей сценарий, а не имени файла, жестко записанного в сценарии, можно создавать ссылки на сценарий или переименовывать его файл, получая при этом из сообщений об ошибках и о порядке использования вполне достоверную информацию.

Присвоение имен временным файлам

Другая особенность сценария safedit связана с использованием параметра $$ в имени временного файла. Инструкция, которая следует за инструкцией esac, создает переменную tempfile и присваивает ей значение. Эта переменная содержит имя временного файла, сохраняемого как и многие другие временные файлы в каталоге /tmp. Имя временного файла начинается с PID-номера оболочки и заканчивается именем сценария.

Использование PID-номера гарантирует уникальность имени файла. Поэтому сценарий safedit не будет предпринимать попытки переписывания существующего файла, что может случиться при одновременном использовании сценария safedit двумя пользователями. Добавление к этому имени названия сценария обусловлено желанием определить источник возникновения файла, если он по каким-либо причинам останется в каталоге /tmp. Расположение PID-номера в имени файла впереди, а не после значения переменной `$script` обусловлено тем, что некоторые старые версии UNIX накладывают на имена файлов 14-символьное ограничение. А системы Linux таких ограничений не накладывают. Поскольку PID-номер гарантирует уникальность имени файла, он ставится впереди и не может быть урезан. (Если будет сокращен компонент, состоящий из значения переменной `$script`, имя файла не утратит своей уникальности.) По той же причине, когда несколькими строками ниже управляющей структурой if создается резервный файл, имя файла содержит строку `bak.`, за которой следует имя редактируемого файла. Если на устаревших системах `bak` использовался бы в качестве суффикса, а не в качестве префикса, а исходное имя файла имело бы длину 14 символов, элемент `.bak` мог бы быть потерян, что привело бы к перезаписи исходного файла. Утилита basename извлекает простое имя файла из значения переменной `$editfile` еще до того, как оно получает префикс `bak`.

Сценарий safedit использует в структуре if весьма необычную *тест-команду*: `vim $editfile`. Эта *тест-команда* вызывает vim для редактирования файла, чье имя является значением переменной `$editfile`. Когда редактирование завершается и происходит выход из редактора vim, этот редактор возвращает код выхода. Управляющая структура if использует этот код выхода для определения, какую ветвь выполнять. Если сеанс редактирования завершился успешно, vim возвращает 0, и выполняется инструкция, которая следует за инструкцией then. Если работа vim завершилась аварийно (что может произойти, если пользователь прекратил выполнение процесса, в котором был запущен редактор vim с помощью утилиты kill [см. часть V]), vim возвращает ненулевой статус выхода, и сценарий выполняет инструкцию, которая следует за инструкцией else.

Управляющая структура select

Управляющая структура select (отсутствующая в tcsh) основана на структуре, имеющейся в оболочке Korn Shell. Она отображает меню, присваивает значение переменной на основе выбранного пользователем пункта меню и выполняет набор команд. Для управляющей структуры select используется следующий синтаксис:

```
select имя_переменной [in аргументы . . . ]
do
    команды
done
```

Структура select выводит меню из элементов *аргументы*. Если опустить ключевое слово in и список аргументов, структура select использует вместо элементов *аргументы* позиционные параметры. Меню форматируется с помощью номеров перед каждым элементом. Например, структура select, начинающаяся с

```
select fruit in apple banana blueberry kiwi orange watermelon STOP
```

выводит следующее меню:

```
1) apple        3) blueberry    5) orange       7) STOP
2) banana       4) kiwi         6) watermelon
```

Структура select использует для размеров отображаемого меню значения переменных `LINES` (строки) и `COLUMNS` (столбцы). (По умолчанию переменная `LINES` имеет

значение 24, а переменная COLUMNS имеет значение 80.) Когда значение переменной COLUMNS установлено в 20, меню приобретает следующий вид:

```
1) apple
2) banana
3) blueberry
4) kiwi
5) orange
6) watermelon
7) STOP
```

PS3

После вывода меню структура select выводит значение переменной PS3, то есть приглашение select. По умолчанию переменная PS3 имеет значение ?#, но обычно для нее устанавливается более осмысленное значение. При вводе в качестве ответа на приглашение PS3 правильного номера (одного из номеров, присутствующих в диапазоне меню) структура select присваивает переменной *имя_переменной* значение аргумента, соответствующее введенному номеру. Неправильный ввод заставляет оболочку установить переменной *имя_переменной* значение нулевой строки. В любом случае структура select сохраняет ваш ответ в переменной REPLY, являющейся ключевым словом, а затем выполняет те команды, которые находятся между инструкциями do и done. Если нажать ВВОД без набора выбранного номера, оболочка повторит вывод меню и приглашения PS3.

Структура select продолжает выдавать приглашение PS3 и выполнять *команды* до тех пор, пока что-нибудь не станет причиной выхода из структуры — обычно для выхода используется инструкция break или exit. Инструкция break осуществляет выход из цикла, а инструкция exit — выход из сценария.

Использование структуры select продемонстрировано в следующем сценарии:

```
$ cat fruit2
#!/bin/bash
PS3="Выберите свой любимый фрукт или ягоду из этого перечня: "
select FRUIT in яблоко банан черника киви апельсин арбуз СТОП
do
    if [ "$FRUIT" == "" ]; then
        echo -e "Неверный ввод.\n"
        continue
    elif [ $FRUIT = STOP ]; then
        echo "Спасибо за игру!"
        break
    fi
echo "Ваш выбор любимого фрукта или ягоды – $FRUIT."
echo -e "Номер выбора $REPLY.\n"
done

$ ./fruit2
1) яблоко       3) черника      5) апельсин       7) СТОП
2) банан        4) киви         6) арбуз
Выберите свой любимый фрукт или ягоду из этого перечня: 3
Ваш выбор любимого фрукта или ягоды – черника.
Номер выбора 3.
```

```
Выберите свой любимый фрукт или ягоду из этого перечня: 99
Неверный ввод.
Выберите свой любимый фрукт или ягоду из этого перечня: 7
Спасибо за игру!
```

После установки значения приглашения PS3 и формирования меню с помощью инструкции select сценарий fruit2 выполняет команды между инструкциями do и done. Если пользователь отправляет неверный ввод, оболочка присваивает переменной *имя_переменной* ($FRUIT) в качестве значения нулевую строку. Если переменная $FRUIT содержит нулевую строку, команда echo выводит сообщение об ошибке, затем инструкция continue заставляет оболочку повторить вывод приглашения PS3. Если ввод имеет допустимое значение, сценарий проверяет, не захотел ли пользователь остановить цикл. Если он захотел это сделать, команда echo выводит сообщение, и инструкция break осуществляет выход из структуры select (и из сценария). Если пользователь ввел допустимый ответ и не хочет останавливать цикл, сценарий выводит название и номер, соответствующие ответу пользователя. (Сведения о ключе -e в команде echo даны выше, сразу после кода сценария command_menu.)

Структура Here Document

Структура Here document (здесь документ) позволяет перенаправить ввод в сценарий оболочки непосредственно из самого сценария оболочки. Свое название структура Here document получила потому, что слово «here» означает здесь, а не там, возможно, в другом файле. В следующем сценарии по имени birthday содержится структура Here document. О том, что далее следует структура Here document, свидетельствуют два символа «меньше чем» (<<) в первой строке. За символами «меньше чем» следует один или несколько символов, определяющих границы структуры Here document, в данном примере для этого используется знак плюс. Тогда как открывающий ограничитель должен примыкать к символам «меньше чем», закрывающий ограничитель должен быть единственным символом в строке. Оболочка отправляет все, что находится между двумя ограничителями, на стандартный ввод процесса. В примере все выглядит так, как будто стандартный ввод перенаправлен в команду grep из файла, за исключением того, что файл встроен в сценарий оболочки:

```
$ cat birthday
grep -i "$1" <<+
Max     June 22
Barbara February 3
Darlene May 8
Helen   March 13
Zach    January 23
Nancy   June 26
+
$ ./birthday Zach
Zach    January 23
$ ./birthday june
Max     June 22
Nancy   June 26
```

При запуске сценария birthday он выводит все строки структуры Here document, которые содержащие аргумент, с которым он был вызван. В данном случае, при первом запуске, сценарий birthday выводит день рождения пользователя Zach, потому что он был вызван с аргументом Zach. При втором запуске он выводит все июньские дни рождения. Аргумент -i заставляет команду grep вести поиск, нечувствительный к регистру букв.

Следующий сценарий по имени bundle[1] включает весьма разумное использование структуры Here document. Сценарий bundle является красивым примером сценария, создающего файл архива оболочки — shell archive (shar). Сценарий создает файл, который сам по себе является сценарием оболочки, содержащим ряд других файлов, а также код для воссоздания исходных файлов:

```
$ cat bundle
#!/bin/bash
# bundle:  группа файлов дистрибуционного пакета

echo "# Для извлечения файлов пакета запустите этот файл в bash"
for i
do
        echo "echo $i 1>&2"
        echo "cat >$i <<'Граница $i'"
        cat $i
        echo "Граница $i"
done
```

Подобно тому, как оболочка не рассматривает в качестве специальных те специальные символы, которые появляются в стандартном вводе сценария оболочки, она не рассматривает в качестве специальных и те специальные символы, которые появляются между ограничителями структуры Here document.

Как показано в следующем примере, выводом сценария bundle (имя которого означает «пакет») является сценарий оболочки, который перенаправляется в файл по имени bothfiles (что означает «оба файла»). В нем внутри структуры Here document находится содержимое каждого файла, имя которого дано сценарию bundle в качестве аргумента (в данном случае это file1 и file2). Для извлечения исходных файлов из файла bothfiles нужно просто дать это имя в качестве аргумента команде bash. Перед каждой структурой Here document сценарий вызывает команду cat, которая при запуске bothfiles заставляет структуру Here document записываться в новый файл:

```
$ cat file1
Это файл.
Он содержит две строки.
$ cat file2
Это еще один файл.
Он содержит
три строки.
$ ./bundle file1 file2 > bothfiles
$ cat bothfiles
# Для извлечения файлов пакета запустите этот файл в bash
echo file1 1>&2
cat >file1 <<'Граница file1'
Это файл.
```

[1] Спасибо Брайану Кернигану (Brian W. Kernighan) и Робу Пайку (Rob Pike), авторам книги «The Unix Programming Environment» (Englewood Cliffs, N.J.: Prentice-Hall, 1984), с. 98. Перепечатка осуществлена с их позволения.

```
Он содержит две строки.
Граница file1
echo file2 1>&2
cat >file2 <<'Граница file2'
Это еще один файл.
Он содержит
три строки.
Граница file2
```

В следующем примере file1 и file2 перед запуском сценария bothfiles удаляются. Сценарий bothfiles с помощью команды echo выводит имена создаваемых им файлов по мере их создания. Затем команда ls показывает, что сценарий bothfiles воссоздал file1 и file2:

```
$ rm file1 file2
$ bash bothfiles
file1
file2
$ ls
bothfiles
file1
file2
```

Описатели файлов

Как говорилось в разделе «Описатели файлов» на с. 289, перед тем как процесс сможет читать файл или вести в него запись, он должен открыть этот файл. Когда процесс открывает файл, Linux связывает с этим файлом определенный номер (который называется *описателем файла*). Описатель файла — это индекс в таблице открытых файлов процесса. Каждый процесс имеет свой собственный набор открытых файлов и свои собственные описатели файлов. После открытия файла процесс проводит чтение этого файла или запись в него, ссылаясь на присвоенный ему описатель файла. Когда файл больше не нужен, процесс его закрывает, освобождая описатель файла. Обычный процесс Linux начинает свою работу с тремя открытыми файлами: стандартным вводом (описатель файла 0), стандартным выводом (описатель файла 1) и стандартной ошибкой (описатель файла 2). Зачастую процессу нужны только эти файлы. Вспомните, что перенаправление стандартного вывода осуществлялось с помощью символа > или символов 1> и что перенаправление стандартной ошибки осуществлялось с помощью символов 2>.

Хотя другие описатели файлов также можно перенаправить, но, поскольку описатели файлов, отличные от 0, 1 и 2, не имеют никакого специально установленного значения, вряд ли это принесет какую-нибудь пользу. Исключение составляет самостоятельно написанная программа, в которой вы управляете значениями описателей файлов и получаете какую-то выгоду от перенаправления.

Открытие описателя файла

Оболочка Bourne Again Shell открывает файлы благодаря использованию встроенной команды exec:

```
exec n> файл_вывода
exec m< файл_ввода
```

Первая строка открывает *файл_вывода* для вывода и держит его открытым, связывая с ним описатель файла *n*. Вторая строка открывает *файл_ввода* для ввода и держит его открытым, связывая с ним описатель файла *m*.

Дублирование описателя файла

Лексема <& дублирует описатель *файла_ввода*, а лексема >& дублирует описатель *файла_вывода*. Можно продублировать описатель файла, сделав его ссылкой на тот же файл, на который ссылается другой открытый описатель файла, например стандартный ввод или вывод. Для открытия или перенаправления описателя файла *n* в качестве дубликата описателя файла *m* используется следующий формат:

exec n<&m

Как только файл будет открыт, его можно использовать для ввода и вывода двумя способами. Во-первых, можно воспользоваться перенаправлением ввода-вывода в любой командной строке, перенаправить стандартный вывод в описатель файла, используя формат >&*n*, или перенаправить стандартный ввод из описателя файла с помощью формата <&*n*. Во-вторых, можно использовать встроенные команды read (см. раздел «read: Получение пользовательского ввода» на с. 466) и echo. При вызове других команд, включая функции (см. раздел «Функции» на с. 363), они наследуют эти открытые файлы и описатели файлов. Когда использование файла завершится, его можно будет закрыть, используя команду

exec n<&-

Когда в следующем примере, который называется mycp, вызывается функция оболочки с двумя аргументами, она копирует файл, имя которого задано в первом аргументе, в файл, имя которого задано во втором аргументе. Если будет предоставлен только один аргумент, файл, имя которого задано в аргументе, будет скопирован в стандартный вывод. При вызове без аргументов mycp скопирует стандартный ввод в стандартный вывод.

ФУНКЦИЯ НЕ ЯВЛЯЕТСЯ СЦЕНАРИЕМ ОБОЛОЧКИ

Пример mycp является функцией оболочки; он не будет работать как от него ожидается, если его выполнить как сценарий оболочки. (Он сработает следующим образом: функция будет создана в подчиненной оболочке с очень коротким периодом существования, в течение которого вы, скорее всего, ею не воспользуетесь.) Эту функцию можно ввести с клавиатуры. Если поместить функцию в файле, его можно будет запустить в качестве аргумента встроенной команды . (точка) (см. раздел «Утилита . (точка) или source: Запуск файла запуска в текущей оболочке» на с. 287). Если нужно, чтобы функция была доступна всегда, ее можно также поместить в файл запуска (см. раздел «Функции в файлах запуска» на с. 345).

```
function mycp ()
{
case $# in
    0)
        # Нуль аргументов
        # Описатель файла 3 дублирует стандартный ввод
        # Описатель файла 4 дублирует стандартный вывод
        exec 3<&0 4<&1
        ;;
    1)
        # Один аргумент
        # Открытие для ввода файла, имя которого задано аргументом,
        # и связывание его с описателем файла 3
```

```
        # Описатель файла 4 дублирует стандартный вывод
        exec 3< $1 4<&1
        ::
    2)
        # Два аргумента
        # Открытие для ввода файла, имя которого задано первым аргументом,
        # и связывание его с описателем файла 3
        # Открытие для вывода файла, имя которого задано вторым аргументом,
        # и связывание его с описателем файла 4
        exec 3< $1 4> $2
        ::
    *)
        echo "Порядок использования: mycp [источник [получатель]]"
        return 1
        ::
esac

# Вызов команды cat с вводом, поступающим из описателя файла 3,
# и выводом, направляемым в описатель файла 4
cat <&3 >&4

# Закрытие описателей файлов 3 и 4
exec 3<&- 4<&-
}
```

Реальная работа этой функции делается в строке, которая начинается с команды cat. Весь остальной код занимается настройкой описателей файлов 3 и 4, которые являются вводом и выводом команды cat, под связь с соответствующими файлами.

Следующая программа получает из командной строки имена двух файлов, сортирует оба файла и отправляет вывод во временный файл. Затем программа объединяет отсортированные файлы в стандартном выводе, предваряя каждую строку номером, показывающим, из какого файла она поступила.

```
$ cat sortmerg
#!/bin/bash
usage ()
{
if [ $# -ne 2 ]; then
      echo "Порядок использования: $0 file1 file2" 2>&1
      exit 1
fi
}

# Временный каталог, используемый по умолчанию
: ${TEMPDIR:=/tmp}
# Проверка счетчика аргументов
usage "$@"
# Задание временных файлов для сортировки
file1=$TEMPDIR/$$.file1
file2=$TEMPDIR/$$.file2
# Сортировка
sort $1 > $file1
sort $2 > $file2
```

```
# Открытие $file1 и $file2 для чтения. Использование описателей файлов 3 и 4.
exec 3<$file1
exec 4<$file2
# чтение первой строки из каждого файла для определения, с чего начать.
read Line1 <&3
status1=$?
read Line2 <&4
status2=$?
# Стратегия: пока ввод осуществляется из обоих файлов:
#   Вывод строки, которая должна следовать первой.
#   Чтение новой строки из того файла, откуда поступила строка.
while [ $status1 -eq 0 -a $status2 -eq 0 ]
    do
        if [[ "$Line2" > "$Line1" ]]; then
            echo -e "1.\t$Line1"
            read -u3 Line1
            status1=$?
        else
            echo -e "2.\t$Line2"
            read -u4 Line2
            status2=$?
        fi
done
# Теперь указатель одного из файлов добрался до конца файла.
# Чтение из каждого файла, пока он не закончится.
# Сначала из file1:
while [ $status1 -eq 0 ]
    do
        echo -e "1.\t$Line1"
        read Line1 <&3
        status1=$?
    done
# Затем из file2:
while [[ $status2 -eq 0 ]]
    do
        echo -e "2.\t$Line2"
        read Line2 <&4
        status2=$?
    done

# Закрытие и удаление обоих файлов ввода
exec 3<&- 4<&-
rm -f $file1 $file2
exit 0
```

Параметры и переменные

Параметры оболочки и переменные были представлены в разделе «Параметры и переменные» на с. 304. В этом разделе прежний материал дополняется рассмотрением переменных, представляющих собой массивы, сравнением глобальных и локальных переменных, специальных и позиционных параметров и расширением нулевых и неустановленных переменных.

Переменные, представляющие собой массивы

Оболочка Bourne Again Shell поддерживает переменные, представляющие собой одномерные массивы. Индексами массивов являются целые числа, а отсчет индексов начинается с нуля (то есть первый элемент массива имеет индекс 0). Объявление массива и присваивание значений его элементам осуществляются с использованием следующего формата:

имя=(элемент1 элемент2 ...)

В следующем примере массиву NAMES присваиваются четыре значения:

```
$ NAMES=(max helen sam zach)
```

Ссылка на отдельный элемент массива производится следующим образом:

```
$ echo ${NAMES[2]}
sam
```

Оба индекса, [*] и [@], приводят к извлечению всего массива, но при использовании внутри двойных кавычек они работают по-разному. Индекс @ выдает массив, являющийся дубликатом исходного массива, а индекс * выдает один элемент массива (или простую переменную), в котором содержатся все элементы массива, отделенные друг от друга первым символом, который находится в переменной IFS (обычно это пробел). В следующем примере массив A заполняется элементами переменной NAMES с использованием индекса *, а массив B заполняется этими элементами с использованием индекса @. Значения массивов выводятся с помощью встроенной команды declare с ключом -a (следует напомнить, что при индексации массивов оболочка bash использует нулевую базу отсчета):

```
$ A=("${NAMES[*]}")
$ B=("${NAMES[@]}")

$ declare -a
declare -a A='([0]="max helen sam zach")'
declare -a B='([0]="max" [1]="helen" [2]="sam" [3]="zach")'
-
declare -a NAMES='([0]="max" [1]="helen" [2]="sam" [3]="zach")'
```

Из вывода команды declare можно увидеть, что массивы NAMES и B имеют несколько элементов. В отличие от них A, которому значение было присвоено с помощью индекса * внутри двойных кавычек, имеет только один элемент: у A все его элементы взяты в двойные кавычки.

В следующем примере команда echo пытается вывести элемент 1 массива A. Но ничего не выводится, поскольку у A есть только один элемент, и этот элемент имеет индекс 0. Элемент 0 массива A содержит все четыре имени. Элемент 1 массива B содержит второй элемент массива, а элемент 0 содержит первый элемент.

```
$ echo ${A[1]}

$ echo ${A[0]}
max helen sam zach
$ echo ${B[1]}
helen
$ echo ${B[0]}
max
```

Для возвращения количества элементов массива к переменным массивов можно применить оператор ${#*имя*[*]}:

```
$ echo ${#NAMES[*]}
4
```

Тот же самый оператор, когда ему вместо * дается индекс элемента массива, возвращает длину этого элемента:

```
$ echo ${#NAMES[1]}
5
```

Чтобы заменить выбранный элемент массива, можно использовать индекс в левой части оператора присваивания:

```
$ NAMES[1]=max
$ echo ${NAMES[*]}
max max sam zach
```

Локальность переменных

По умолчанию переменные являются локальными по отношению к процессу, в котором они были объявлены. Поэтому сценарий оболочки не имеет доступа к переменным, объявленным в вашей оболочке входа в систему до тех пор, пока вы не сделаете переменные доступными (глобальными). При работе под управлением оболочки bash переменные становятся доступны дочерним процессам благодаря команде export. При работе под управлением оболочки tcsh, чтобы сделать переменную доступной для дочерних процессов, нужно присвоить ей значение с помощью команды setenv (см. раздел «Подстановка переменной» на с. 384). В примерах этого раздела используется синтаксис оболочки bash, но теоретические положения применимы к обеим оболочкам.

export Как только встроенная команда export будет вызвана с аргументом, в качестве которого будет имя переменной, оболочка поместит значение этой переменной в среду вызова дочерних процессов. Эта передача по значению дает каждому дочернему процессу для его собственного пользования копию переменной.

Следующий сценарий оболочки extest1 присваивает значение american переменной по имени cheese, а затем выводит имя файла этого сценария (extest1) и значение переменной cheese. После этого сценарий extest1 вызывает сценарий subtest, который пытается вывести ту же самую информацию. После чего subtest объявляет переменную cheese и выводит ее значение. Когда subtest завершает свою работу, он возвращает управление родительскому процессу, который выполняет сценарий extest1. В этот момент extest1 опять выводит значение исходной переменной cheese.

```
$ cat extest1
cheese=american
echo "extest1 1: $cheese"
subtest
echo "extest1 2: $cheese"
$ cat subtest
echo "subtest 1: $cheese"
cheese=swiss
echo "subtest 2: $cheese"
```

```
$ ./extest1
extest1 1: american
subtest 1:
subtest 2: swiss
extest1 2: american
```

Сценарий subtest никогда не получает значения переменной cheese из сценария extest1, и extest1 никогда не теряет этого значения. В bash, в отличие от реальной действительности, дети не в состоянии повлиять на свойства родителей. Когда процесс пытается вывести значение необъявленной переменной, как в данном случае со сценарием subtest, на экран ничего не выводится, а значением необъявленной переменной служит нулевая строка.

Следующий сценарий extest2 похож на extest1, за исключением того, что в нем используется команда export, чтобы сделать переменную cheese доступной сценарию subtest:

```
$ cat extest2
export cheese=american
echo "extest2 1: $cheese"
subtest
echo "extest2 2: $cheese"
$ ./extest2
extest2 1: american
subtest 1: american
subtest 2: swiss
extest2 2: american
```

Здесь дочерний процесс наследует значение переменной cheese как american и после его вывода изменяет значение *своей копии* на swiss. Когда управление возвращается родителю, родительская копия переменной cheese сохраняет ее исходное значение:

american.

Встроенная команда export может дополнительно включать оператор присваивания:

```
export cheese=american
```

Предыдущая инструкция является эквивалентом следующих двух инструкций:

```
cheese=american
export cheese
```

Хотя это делается довольно редко, переменную можно экспортировать до того, как ей присваивается значение. После изменения значения ранее экспортированной переменной осуществлять ее повторный экспорт нет необходимости.

Функции

Поскольку функции работают в той же среде окружения, что и оболочка, которая их вызвала, переменные неявным образом совместно используются оболочкой и той функцией, которую она вызвала.

```
$ function nam () {
> echo $myname
> myname=zach
> }
```

```
$ myname=sam
$ nam
sam
$ echo $myname
zach
```

В предыдущем примере переменной myname в интерактивной оболочке было присвоено значение sam. Затем функция nam вывела значение переменной myname (sam) и установила для нее значение zach. Заключительная команда echo показала, что в интерактивной оболочке значение переменной myname изменилось на zach.

Локальные переменные функции

Локальные переменные целесообразно применять в функции, написанной для общего пользования. Поскольку функция вызывается многими сценариями, которые могут быть написаны разными программистами, нужно не допустить конфликта имен переменных, используемых внутри функции (то есть дублирования), с именами переменных в программе, которая вызывает функцию. Локальные переменные устраняют эту проблему. При использовании внутри функции встроенная команда typeset объявляет переменную локальной по отношению к функции, в которой она определяется.

В следующем примере показано использование локальной переменной в функции. В нем фигурируют две переменные по имени count. Первая из них объявлена и получила значение 10 в интерактивной оболочке. Ее значение никогда не меняется, о чем свидетельствует вывод команды echo, вызванной после запуска функции count_down. Другая переменная count объявлена с помощью команды typeset локальной по отношению к функции. Ее значение, которое неизвестно за пределами функции, изменяется в диапазоне от 4 до 1, что подтверждается выводом команды echo, вызываемой внутри функции. В примере показана функция, введенная с клавиатуры, которая не является сценарием оболочки (см. совет «Функция не является сценарием оболочки» на с. 450).

```
$ function count_down () {
> typeset count
> count=$1
> while [ $count -gt 0 ]
> do
> echo "$count…"
> ((count=count-1))
> sleep 1
> done
> echo "Пуск."
> }
$ count=10
$ count_down 4
4…
3…
2…
1…
Пуск.
$ echo $count
10
```

Присваивание `((count=count-1))` взято в двойные круглые скобки, которые заставляют оболочку осуществить арифметическое вычисление (см. раздел «Арифметические вычисления» на с. 480). Внутри двойных круглых скобок можно ссылаться на переменные оболочки без лидирующего знака доллара (`$`).

Специальные параметры

Специальные параметры позволяют получить доступ к полезным переменным, имеющим отношение к аргументам командной строки и выполнению команд оболочки. Ссылка на специальные параметры оболочки осуществляется путем установки перед специальным символом знака доллара (`$`). Значение специального параметра, так же как и значение позиционного параметра, невозможно изменить путем присваивания ему значения.

$$: PID-номер

Оболочка сохраняет в параметре `$$` PID-номер процесса, в котором она выполняется. В следующем интерактивном действии команда **echo** выводит значение этой переменной, а утилита **ps** подтверждает ее значение. Обе команды показывают, что у оболочки имеется PID-номер 5209:

```
$ echo $$
5209
$ ps
  PID TTY          TIME CMD
 5209 pts/1    00:00:00 bash
 6015 pts/1    00:00:00 ps
```

Поскольку команда **echo** встроена в оболочку, то при вводе этой команды оболочка не создает для нее еще один процесс. Но результат все равно будет один и тот же, независимо от того, встроена команда **echo** или нет, потому что оболочка осуществляет подстановку значения параметра `$$` перед тем, как она ответвляет новый процесс для запуска команды. Попробуйте воспользоваться утилитой **echo** (**/bin/echo**), которая запускается другим процессом, и посмотрите, что из этого получится. В следующем примере оболочка осуществляет подстановку значения параметра `$$` и передает это значение команде `cp` в качестве префикса имени файла:

```
$ echo $$
8232
$ cp memo $$.memo
$ ls
8232.memo memo
```

Включение PID-номера в имя файла подходит для создания уникальных имен файлов, когда значение имени не играет никакой роли; эта технология часто применяется в сценариях оболочки для создания имен временных файлов. Когда два человека запускают один и тот же сценарий оболочки, наличие уникальных имен файлов избавляет пользователей от непреднамеренного совместного использования одного и того же временного файла.

В следующем примере показано, как оболочка создает новый процесс оболочки при запуске сценария. Сценарий id2 выводит PID-номер процесса, в котором он запущен (а не процесса, который его вызвал; подстановка параметра $$ осуществляется той оболочкой, которая была ответвлена для запуска сценария id2):

```
$ cat id2
echo "$0 PID= $$"
$ echo $$
8232
$ id2
./id2 PID= 8362
$ echo $$
8232
```

Первая команда echo выводит PID-номер интерактивной оболочки. Затем сценарий id2 выводит свое имя ($0) и PID-номер подчиненной оболочки, в которой он запущен. Последняя команда echo показывает, что PID-номер интерактивной оболочки не изменился.

$! Оболочка сохраняет значение PID-номера последнего процесса, который был запущен в фоновом режиме, в параметре $! (который не доступен в tcsh). В следующем примере команда sleep выполняется в качестве фоновой задачи, а команда echo используется для вывода значения параметра $!:

```
$ sleep 60 &
[1] 8376
$ echo $!
8376
```

$?: Статус выхода

Когда процесс по какой-то причине останавливается, он возвращает своему родительскому процессу *статус выхода*. Этот статус еще называют *кодом условия* или *кодом завершения*. Параметр $? (в tcsh параметр $status) сохраняет статус выхода последней команды.

По соглашению ненулевой статус выхода представляет собой значение *false* и означает, что команда завершилась неудачно. А нуль представляет собой *true* и означает, что выполнение команды завершилось успешно. В следующем примере первая команда ls выполняется успешно, а вторая терпит неудачу, о чем свидетельствует статус выхода:

```
$ ls es
es
$ echo $?
0
$ ls xxx
ls: xxx: No such file or directory  Файл или каталог с таким именем не существует
$ echo $?
1
```

Указать статус выхода, возвращаемый сценарием оболочки, можно с помощью встроенной команды exit, предназначенной для завершения сценария, за которой следует число. Если для завершения сценария не используется команда exit с числом,

то статус выхода из сценария совпадает со статусом выхода из последней, запущенной в нем команды.

```
$ cat es
echo Эта программа вернула статус выхода 7.
exit 7
$ es
Эта программа вернула статус выхода 7.
$ echo $?
7
$ echo $?
0
```

Сценарий оболочки es выводит сообщение и завершает выполнение с помощью команды exit, возвращающей статус выхода 7, определенный пользователем статус выхода из сценария. Затем первая команда echo выводит значение статуса выхода из сценария es. А вторая команда echo выводит значение статуса выхода из первой команды echo. Это значение равно 0 и свидетельствует о том, что первая команда echo завершилась успешно.

Позиционные параметры

Позиционные параметры содержат имя команды и аргументы командной строки. Эти параметры называются позиционными, потому что внутри сценария оболочки ссылка на них осуществляется по их позиции в командной строке. Изменять значения позиционных параметров позволяет лишь встроенная команда set (см. раздел «set: Инициализация аргументов командной строки» на с. 461). Но изменить имя команды из сценария невозможно. Встроенная команда set оболочки tcsh значений позиционных параметров не изменяет.

$#: Количество аргументов командной строки

Параметр $# содержит количество аргументов командной строки (позиционных параметров), не включающее саму команду:

```
$ cat num_args
echo "Этот сценарий был вызван с количеством аргументов, равным $#."
$ ./num_args sam max zach
Этот сценарий был вызван с количеством аргументов, равным 3
```

$0: Имя вызванной программы

Оболочка сохраняет имя команды, которая была использована для вызова программы в параметре $0. Этот параметр имеет нулевой номер, потому что он появляется перед первым аргументом командной строки:

```
$ cat abc
echo "Для запуска этого сценария использовалась команда $0"
$ ./abc
Для запуска этого сценария использовалась команда ./abc
$ ~sam/abc
Для запуска этого сценария использовалась команда /home/sam/abc
```

Предыдущий сценарий оболочки использует команду echo для подтверждения имени выполняемого сценария. Для извлечения и вывода простого имени файла команды можно воспользоваться утилитой basename и командной подстановкой:

```
$ cat abc2
echo " Для запуска этого сценария использовалась команда $(basename $0)"
$ ~sam/abc2
Для запуска этого сценария использовалась команда abc2
```

$1–$n: Аргументы командной строки

Первый аргумент командной строки представлен параметром $1, второй аргумент – параметром $2 и так далее, вплоть до $*n*. Для значений *n* больше 9 номер должен быть взят в фигурные скобки. Например, двенадцатый аргумент командной строки представлен параметром ${12}. Следующий сценарий выводит позиционные параметры, которые содержат аргументы командной строки:

```
$ cat display_5args
echo Первые пять аргументов $1 $2 $3 $4 $5
$ ./display_5args zach max helen
Первые пять аргументов zach max helen
```

Сценарий display_5args выводит первые пять аргументов командной строки. Оболочка присваивает нулевое значение каждому параметру, представляющему аргумент, отсутствующий в командной строке. Поэтому параметры $4 и $5 имеют в данном примере нулевые значения.

shift: Смещение аргументов командной строки

Встроенная команда shift смещает каждый аргумент командной строки. Первый аргумент (которому соответствовал параметр $1) удаляется. Второй аргумент (которому соответствовал параметр $2) становится первым (теперь ему соответствует параметр $1), третий становится вторым и т. д. Поскольку обратной по действию команды «unshift» не существует, восстановить удаленные аргументы уже невозможно. Дополнительный аргумент команды shift указывает количество позиций сдвига (и количество удаляемых аргументов); по умолчанию это количество равно 1.

Следующий сценарий demo_shift вызывается с тремя аргументами. Двойные кавычки вокруг аргументов предназначены для того, чтобы команда echo сохраняла пробелы в своем выводе. Программа выводит аргументы и смещает их несколько раз, пока не останется ни одного аргумента для смещения:

```
$ cat demo_shift
echo "arg1= $1    arg2= $2    arg3= $3"
shift
echo "arg1= $1    arg2= $2    arg3= $3"
shift
echo "arg1= $1    arg2= $2    arg3= $3"
shift
echo "arg1= $1    arg2= $2    arg3= $3"
shift
$ ./demo_shift alice helen zach
```

```
arg1= alice    arg2= helen    arg3= zach
arg1= helen    arg2= zach     arg3=
arg1= zach     arg2=    arg3=
arg1=     arg2=    arg3=
```

Многократное использование команды shift является удобным способом последовательного перебора всех аргументов командной строки в сценарии оболочки, который ожидает произвольное количество аргументов. Сценарий, использующий команду shift, показан в разделе «Управляющая структура if...then...else» на с. 421.

set: Инициализация аргументов командной строки

Когда встроенная команда set вызывается с одним и более аргументами, она присваивает значения аргументов позиционным параметрам начиная с $1 (в tcsh такая операция недоступна). Следующий сценарий использует команду set для присваивания значений позиционным параметрам $1, $2 и $3:

```
$ cat set_it
set this is it
echo $3 $2 $1
$ ./set_it
it is this
```

Сочетание командной подстановки (см. раздел «Командная подстановка» на с. 358) со встроенной командой set является удобным способом получения стандартного вывода команды в форме, которой можно легко управлять в сценарии оболочки. В следующем сценарии показано, как использовать команды date и set для предоставления даты в удобном формате. Первая команда показывает вывод даты. Затем команда cat выводит содержимое сценария dateset. Первая команда в этом сценарии использует командную подстановку для установки значения позиционного параметра на вывод утилиты date. Следующая команда, echo $*, выводит все позиционные параметры, полученные благодаря вызову предыдущей команды set. Следующие команды выводят значения параметров $1, $2, $3 и $6. Последняя команда выводит дату в формате, который можно использовать в письме или отчете:

```
$ date
Wed Aug 13 17:35:29 PDT 2008
$ cat dateset
set $(date)
echo $*
echo
echo "Аргумент 1: $1"
echo "Аргумент 2: $2"
echo "Аргумент 3: $3"
echo "Аргумент 6: $6"
echo
echo "$2 $3, $6"
$ ./dateset
Wed Aug 13 17:35:34 PDT 2008
Аргумент 1: Wed
Аргумент 2: Aug
Аргумент 3: 13
```

```
Аргумент 6: 2008
Aug 13, 2008
```

Для изменения формата вывода команды date можно также воспользоваться аргументом *+формат* (см. описание утилиты date в части V).

При использовании без аргументов команда set выводит список установленных переменных оболочки, включая переменные, созданные пользователем, и переменные, являющиеся ключевыми словами. Под управлением bash этот список будет таким же, как и тот, что выводится командами declare и typeset, когда они вызываются без аргументов.

Встроенная команда set также воспринимает ключи, позволяющие настроить поведение оболочки (в tcsh эта операция недоступна). Дополнительные сведения даны в разделе «set ±o: Включение и выключение свойств оболочки» на с. 348.

$* и $@: Представление всех аргументов командной строки

Как показано в программе display_all, параметр $* представляет все аргументы командной строки:

```
$ cat display_all
echo Все аргументы: $*
$ ./display_all a b c d e f g h i j k l m n o p
Все аргументы: a b c d e f g h i j k l m n o p
```

Заключение ссылок на позиционные параметры в двойные кавычки имеет вполне определенную практическую пользу. Кавычки играют особо важную роль, когда позиционные параметры используются в качестве аргументов команды. Без двойных кавычек позиционный параметр, который не был установлен командой set или который имел нулевое значение, просто исчезает:

```
$ cat showargs
echo "При вызове $0 было использовано аргументов: $#, первый аргумент:$1:."

$ ./showargs a b c
При вызове ./showargs было использовано аргументов: 3, первый аргумент:a:.
$ echo $xx

$ ./showargs $xx a b c
При вызове ./showargs было использовано аргументов: 3, первый аргумент:a:.
$ ./showargs «$xx" a b c
При вызове ./showargs было использовано аргументов: 4, первый аргумент::.
```

Сценарий showargs выводит количество аргументов ($#), за которым следует первый аргумент, взятый с обеих сторон в двоеточия. В предыдущем примере сценарий showargs сначала был вызван с тремя простыми аргументами. Затем команда echo показала, что неустановленная переменная $xx имеет нулевое значение. При последних двух вызовах сценария showargs в качестве первого аргумента была использована переменная $xx. В первом случае командная строка имела вид showargs a b c, и оболочка передала сценарию showargs три аргумента. Во втором случае командная строка приобрела вид showargs "" a b c, в результате чего сценарий showargs был вызван с четырьмя

аргументами. Разница между этими двумя вызовами showargs иллюстрирует едва уловимую потенциальную проблему, о которой нужно помнить при использовании неустановленных или нулевых позиционных параметров.

Сравнение "$*" и "$@"

Параметры $* и $@ работают одинаково, за исключением того случая, когда они заключены в двойные кавычки. Использование параметра "$*" приводит к получению одного аргумента (с пробелом или значением первого символа переменной IFS [см. главу 8, раздел «Переменная IFS: Разделение полей ввода (разбиение слов)» на с. 316] между позиционными параметрами), а использование параметра "$@" приводит к получению списка, в котором каждый позиционный параметр является отдельным аргументом.

Эта разница делает параметр "$@" более полезным для применения в сценариях оболочки, чем параметр "$*". Следующие сценарии помогают объяснить разницу между этими двумя специальными параметрами. Во вторых строках обоих сценариев одинарные кавычки не дают оболочке интерпретировать взятые в них специальные символы, поэтому они передаются команде echo и выводятся в своем непосредственном виде. Сценарий bb1 показывает, что команда set "$*" присваивает несколько аргументов первому параметру командной строки:

```
$ cat bb1
set "$*"
echo Количество параметров с '"$*"' - $#
echo 1: $1
echo 2: $2
echo 3: $3
$ ./bb1 a b c
Количество параметров с "$*" - 1
1: a b c
2:
3:
```

Сценарий bb2 показывает, что команда set "$@" присваивает каждый аргумент разному параметру командной строки:

```
$ cat bb2
set "$@"
echo Количество параметров с '"$@"' - $#
echo 1: $1
echo 2: $2
echo 3: $3
$ ./bb2 a b c
Количество параметров с "$@" - 3
1: a
2: b
3: c
```

Расширение нулевых и неустановленных переменных

Выражение ${*имя*} (или просто $*имя*, если не возникает двояких толкований) расширяется в значение переменной *имя*. Если переменная *имя* имеет нулевое значение или ее значение не установлено, bash расширяет выражение ${*имя*} в нулевую строку.

Вместо принятия нулевой строки в качестве значения переменной оболочка Bourne Again Shell предоставляет следующие альтернативные варианты:

- Использование для переменной значения по умолчанию.
- Использование значения по умолчанию и присваивание этого значения переменной.
- Вывод сообщения об ошибке.

Один из этих вариантов можно выбрать с помощью использования модификатора с именем переменной. Кроме того, чтобы заставить оболочку bash выводить сообщение об ошибке и выходить из сценария при ссылке на неустановленную переменную, можно воспользоваться командой `set -o nounset` (см. табл. 8.13).

:– Использование значения по умолчанию

Модификатор :- использует вместо нулевой или неустановленной переменной значение по умолчанию, позволяя ненулевой переменной представлять саму себя:

${имя:–значение_по_умолчанию}

Оболочка интерпретирует модификатор :- следующим образом: «Если переменная *имя* нулевая или неустановленная, расширить *значение_по_умолчанию* и использовать расширенное значение вместо *имени*». Следующая команда выводит список каталогов, указанных в переменной LIT. Если эта переменная имеет нулевое или неустановленное значение, команда выводит список содержимого каталога /home/max/literature:

```
$ ls ${LIT:-/home/max/literature}
```

Расширяемые ссылки на переменные могут содержаться и в самом *значении_по_умолчанию*:

```
$ ls ${LIT:-$HOME/literature}
```

:= Присваивание значения по умолчанию

Модификатор :- не изменяет значения переменной. Но в сценарии значение нулевой или неустановленной переменной можно изменить на ее значение по умолчанию с помощью модификатора :=

${имя:=значение_по_умолчанию}

Оболочка расширяет выражение *${имя:=значение_по_умолчанию}* точно так же, как она расширяет *${имя:-значение_по_умолчанию}*, но кроме этого она присваивает переменной *имя* расширенное *значение_по_умолчанию*. Если сценарий содержит такую же строку, как следующая, и переменная LIT на момент выполнения этой строки имеет нулевое или неустановленное значение, переменной LIT присваивается значение /home/max/literature:

```
$ ls ${LIT:=/home/max/literature}
```

Встроенная команда : (нулевая строка)

Сценарии оболочки часто начинаются со встроенной команды : (нулевая строка), за которой в той же строке следует модификатор расширения :=, предназначенный для установки значений любых нулевых или неустановленных переменных. Встроенная команда : вычисляет каж-

дую лексему в остальной части командной строки, но не выполняет никаких команд. Без лидирующего двоеточия (:) оболочка вычисляет и пытается выполнить «команду», полученную в результате вычисления.

Для установки значения по умолчанию для нулевой или неустановленной переменной в сценарии оболочки используется следующий синтаксис (за первым двоеточием следует пробел):

: ${имя:=значение_по_умолчанию}

Когда сценарию нужен каталог для временных файлов и для получения имени этого каталога используется значение переменной TEMPDIR, следующая строка присваивает переменной TEMPDIR значение /tmp, если TEMPDIR имеет нулевое значение:

```
: ${TEMPDIR:=/tmp}
```

:? Вывод сообщения об ошибке

Иногда сценарии нуждаются в значении переменной, но на момент написания сценария вы не можете предоставит ей какого-нибудь приемлемого значения по умолчанию. Если переменная имеет нулевое значение или она не установлена, модификатор :? заставляет сценарий вывести сообщение об ошибке и завершить сценарий со статусом выхода 1:

${имя:?сообщение}

Если опустить *сообщение*, оболочка выведет сообщение об ошибке, используемое по умолчанию (о том что параметр имеет нулевое значение или не установлен — «parameter null or not set»). При использовании модификатора :? в интерактивной оболочке выхода из оболочки не происходит. В следующей команде переменная TESTDIR не установлена, поэтому оболочка выводит в стандартную ошибку расширенное значение строки, которая следует за модификатором :?. В данном случае строка включает командную подстановку для даты в формате %T, за которой следует строка, в которой сообщается: «ошибка, переменная не установлена».

```
cd ${TESTDIR:?$(date +%T) ошибка, переменная не установлена.}
bash: TESTDIR: 16:16:14 ошибка, переменная не установлена.
```

Встроенные команды

Встроенные команды, представленные в главе 5, не вызывают при своем выполнении ответвления нового процесса. В этом разделе рассматриваются встроенные команды type, read, exec, trap, kill и getopts. В табл. 10.6 приведены многие встроенные команды оболочки bash. А встроенные команды оболочки tcsh приведены в табл. 9.10.

type: Вывод информации о команде

Встроенная команда type (в tcsh используется команда which) предоставляет информацию о команде:

```
$ type cat echo who if lt
cat is hashed (/bin/cat)
```

```
echo is a shell builtin
who is /usr/bin/who
if is a shell keyword
lt is aliased to 'ls -ltrh | tail'
```

Предыдущий вывод показывает файлы, которые будут выполнены, если cat или who будут введены в качестве команд. Поскольку cat уже была вызвана из текущей оболочки, она находится в хеш-таблице, и команда type сообщает о том, что команда cat хеширована. В выводе также показано, что вызов команды echo запустит встроенную команду echo, что if является ключевым словом, а команда lt является псевдонимом.

read: Получение пользовательского ввода

Одним из самых распространенных способов использования пользовательских переменных является хранение информации, которую пользователь ввел в ответ на приглашение. Чтение пользовательского ввода под управлением оболочки tcsh рассмотрено в разделе «Чтение пользовательского ввода» на с. 390. Встроенная команда считывает одну строку со стандартного ввода и присваивает слова этой строки одной или нескольким переменным:

```
$ cat read1
echo -n "Введите строку: "
read firstline
echo "Вы ввели: $firstline"
$ ./read1
Введите строку: Это строка.
Вы ввели: Это строка.
```

В первой строке сценария read1 для запроса строки текста используется команда echo. Ключ -n подавляет следующий разделитель строк, позволяя ввести строку текста в ту же самую строку, на которой выведено приглашение. Во второй строке текст считывается в переменную firstline.

В третьей строке проверяется действие команды read путем вывода значения переменной firstline. Переменная в этом примере взята в кавычки (вместе со строкой текста), потому что вы, как создатель сценария, не можете заранее предположить, что может ввести пользователь в ответ на приглашение. Рассмотрим, что случится, если переменная не взята в кавычки и пользователь ввел * в ответ на приглашение:

```
$ cat read1_no_quote
echo -n " Введите строку: "
read firstline
echo Вы ввели: $firstline
$ ./read1_no_quote
Введите строку: *
Вы ввели: read1 read1_no_quote script.1
$ ls
read1   read1_no_quote    script.1
```

Команда ls выводит те же слова, что и сценарий, демонстрируя, что оболочка расширяет звездочку в список файлов в рабочем каталоге. Когда переменная $firstline взята в двойные кавычки, оболочка не расширяет звездочку. Поэтому сценарий read1 работает правильно:

```
$ ./read1
Введите строку: *
Вы ввели: *
```

REPLY Встроенная команда read имеет ряд особенностей, облегчающих ее использование. Например, когда не указана переменная для получения ввода, поступившего команде read, оболочка bash помещает ввод в переменную по имени REPLY. Вместо использования отдельной команды echo, для вывода приглашения пользователю на ввод можно воспользоваться ключом -p. Следующий сценарий read1a делает то же самое, что и сценарий read1:

```
$ cat read1a
read -p "Введите строку: "
echo "Вы ввели: $REPLY"
```

Сценарий read2 выводит приглашение на ввод командной строки, считывает ответ пользователя и присваивает его переменной cmd. Затем сценарий пытается выполнить командную строку, которая получается после расширения переменной cmd:

```
$ cat read2
read -p "Введите команду: " cmd
$cmd
echo "Спасибо"
```

В следующем примере сценарий read2 считывает командную строку, которая вызывает встроенную команду echo. Оболочка выполняет команду, а затем выводит слово «Спасибо». Затем сценарий read2 считывает командную строку, которая вызывает выполнение утилиты who:

```
$ ./read2
Введите команду: echo Пожалуйста, выведи это сообщение.
Пожалуйста, выведи это сообщение.
Спасибо
$ ./read2
Введите команду: who
max      pts/4        Jun 17 07:50  (:0.0)
sam      pts/12       Jun 17 11:54  (bravo.example.com)
Спасибо
```

Если переменная cmd не расширяется в допустимую командную строку, оболочка выводит сообщение об ошибке:

```
$ ./read2
Введите команду: xxx
./read2: line 2: xxx: command not found     команда не найдена
Спасибо
```

Сценарий read3 считывает значения в три переменные. Встроенная команда read присваивает по одному слову (последовательности непустых символов) каждой переменной:

```
$ cat read3
read -p "Введите что-нибудь: " word1 word2 word3
echo "Слово 1: $word1"
echo "Слово 2: $word2"
echo "Слово 3: $word3"
```

```
$ ./read3
Введите что-нибудь: раз два три
Слово 1: раз
Слово 2: два
Слово 3: три
```

При вводе количества слов, превышающего количество переменных, объявленных командой read, эта команда присваивает одно слово каждой переменной, при этом все оставшиеся слова присваиваются последней переменной. Оба сценария read1 и read2 присваивают первое слово и все оставшиеся слова одной переменной сценария, с которой каждому из них приходится работать. В следующем примере команда read присваивает пять слов трем переменным: она присваивает первое слово первой переменной, второе слово второй переменной, а с третьего по пятое слово присваиваются третьей переменной.

```
$ ./read3
Введите что-нибудь: раз два три четыре пять.
Слово 1:  раз
Слово 3:  два
Слово 3:  три четыре пять.
```

В табл. 10.4 перечислены некоторые ключи, поддерживаемые встроенной командой read.

Таблица 10.4. Ключи команды read

Ключ	Функция
`-a` *aname*	(массив) Присваивание каждого слова ввода элементу массива по имени *aname*
`-d` *delim*	(разделитель) Использование *delim* для завершения ввода вместо разделителя строк
`-e`	(Readline) Если ввод поступает с клавиатуры, использование библиотеки Readline (см. раздел «Библиотека Readline на с. 334) для получения ввода
`-n` *num*	(количество символов) Считывание *num* символов и возвращение управления. Как только пользователь набирает *num* символов, команда read возвращает управление; нажимать ВВОД не нужно
`-p` *prompt*	(приглашение) Вывод приглашения в стандартный вывод без его завершения разделителем строк перед чтением ввода. Вывод приглашения только в том случае, когда ввод поступает с клавиатуры
`-s`	(молчание) Выключение дублирования символов на экране

`-u`*n*	(описатель файла) Использование целого числа *n* в качестве описателя файла, из которого команда read получает свой ввод. То есть команда
	`read -u4 arg1 arg2`
	является эквивалентом команды
	`read arg1 arg2 <&4`
	См. раздел «Описатели файлов» на с. 449, в котором рассматриваются перенаправление и описатели файлов

В случае успешного чтения данных, встроенная команда read возвращает статус выхода 0. Ненулевой статус выхода эта команда выдает в случае достижения конца файла — EOF.

В следующем примере из командной строки запускается цикл while. Он принимает ввод из файла names и завершает работу после чтения последней строки из names.

```
$ cat names
Alice Jones
Robert Smith
Alice Paulson
John Q. Public

$ while read first rest
> do
> echo $rest, $first
> done < names
Jones, Alice
Smith, Robert
Paulson, Alice
Q. Public, John
$
```

Для структуры while существенную роль играет место, куда ставится символ перенаправления (<). Этот символ нужно поместить за инструкцией done, а не за вызовом команды read.

При каждом перенаправлении ввода оболочка открывает файл, откуда происходит ввод данных, и перемещает позицию указателя чтения в начало файла:

```
$ read line1 < names; echo $line1; read line2 < names; echo $line2
Alice Jones
Alice Jones
```

Здесь каждая команда read открывает файл names и приступает к чтению с начала файла names. В следующем примере файл names открывается только один раз, в качестве стандартного ввода подчиненной оболочки, созданной с помощью круглых скобок. После этого каждая команда read читает последовательные строки стандартного ввода:

```
$ (read line1; echo $line1; read line2; echo $line2) < names
Alice Jones
Robert Smith
```

Другой способ достижения такого же эффекта заключается в открытии файла ввода с помощью команды exec и в сохранении его в открытом состоянии (см. раздел «Описатели файлов» на с. 449):

```
$ exec 3< names
$ read -u3 line1; echo $line1; read -u3 line2; echo $line2
Alice Jones
Robert Smith
$ exec 3<&-
```

exec: Выполнение команды или перенаправление описателей файлов

Встроенная команда exec (отсутствующая в tcsh) имеет два основных предназначения: запуск команды без создания нового процесса и перенаправление описателя файла, включая стандартный ввод, вывод или ошибку (см. раздел «Описатели файлов» на с. 449). Когда оболочка выполняет команду, не встроенную в саму оболочку, она обычно создает новый процесс. Этот новый процесс наследует переменные окружения (глобальные или экспортированные) от своего родительского процесса, но не

наследует переменные, которые не были экспортированы родительским процессом. (Дополнительные сведения даны в разделе «Локальность переменных» на с. 454.) В отличие от этого команда **exec** выполняет команду вместо (поверх) текущего процесса.

exec: Выполнение команды

Встроенная команда **exec**, используемая для запуска команды, имеет следующий синтаксис:

*exec **команда аргументы***

exec в сравнении с . (точкой)

В том, что команда **exec** запускает команду в среде исходного процесса, она похожа на команду. (точка) (см. раздел «Утилита . (точка) или source: Запуск файла запуска в текущей оболочке» на с. 287). Но в отличие от команды . (точка), которая может запускать только сценарии оболочки, команда **exec** может запускать как сценарии, так и скомпилированные программы. Кроме этого, команда . (точка) по завершении своего выполнения возвращает управление исходному сценарию, а команда **exec** этого не делает. И наконец, команда . (точка) дает новой программе доступ к локальным переменным, а команда **exec** нет.

exec не возвращает управление

Поскольку при использовании команды **exec** оболочка не создает нового процесса, команда запускается быстрее. Но из-за того что команда **exec** не возвращает управление исходной программе, она может быть использована только как последняя команда сценария. Следующий сценарий показывает, что управление не возвращается сценарию:

```
$ cat exec_demo
who
exec date
echo "Эта строка никогда не выводится на экран."

$ ./exec_demo
zach     pts/7    May 30  7:05 (bravo.example.com)
hls      pts/1    May 30  6:59 (:0.0)
Mon May 25 11:42:56 PDT 2009
```

В следующем примере измененная версия сценария **out** (см. раздел «Управляющая структура if...then...else» на с. 401) использует команду **exec** для выполнения последней команды, запускаемой сценарием. Поскольку сценарий **out** запускает либо команду **cat**, либо команду **less**, а затем завершает свою работу, новая версия, названная **out2**, использует **exec** как для **cat**, так и для **less**:

```
$ cat out2
if [ $# -eq 0 ]
    then
        echo "Usage: out2 [-v] filenames" 1>&2
        exit 1
fi
if [ "$1" = "-v" ]
    then
        shift
        exec less "$@"
```

```
   else
      exec cat -- "$@"
fi
```

exec: Перенаправление ввода и вывода

Вторым основным предназначением команды **exec** является перенаправление описателя файла, включая стандартный ввод, вывод или ошибку, из самого сценария. Следующая команда заставляет весь последующий ввод в сценарий, который должен поступать из стандартного ввода, поступать из файла по имени **infile**:

```
exec < infile
```

Точно так же следующая команда перенаправляет стандартный вывод и стандартную ошибку в файлы **outfile** и **errfile** соответственно:

```
exec > outfile 2> errfile
```

При использовании команды **exec** таким образом текущий процесс не заменяется новым процессом, и за командой **exec** могут следовать другие команды сценария.

/dev/tty

При перенаправлении вывода из сценария в файл нужно убедиться в том, что пользователь видит приглашения, выводимые сценарием. Устройство **/dev/tty** является псевдонимом экрана, с которым работает пользователь; это устройство можно использовать для ссылки на пользовательский экран, не зная о том, что это за устройство. (Утилита **tty** выводит имя используемого устройства.) Путем перенаправления вывода из сценария в **/dev/tty** гарантируется, что приглашения и сообщения попадут на пользовательский терминал независимо от того, с какого терминала пользователь вошел в систему. Сообщения, отправляемые на **/dev/tty**, также не переадресовываются при перенаправлении из сценария стандартного вывода и стандартной ошибки.

Сценарий **to_screen1** отправляет вывод в три места: в стандартный вывод, в стандартную ошибку и на экран пользователя. При запуске с перенаправленными стандартным выводом и стандартной ошибкой сценарий **to_screen1** продолжает выводить сообщения, отправленные на **/dev/tty**, на пользовательский экран. Файлы **out** и **err** содержат вывод, отправленный на стандартный вывод и на стандартную ошибку.

```
$ cat to_screen1
echo "сообщение на стандартный вывод"
echo "сообщение на стандартную ошибку" 1>&2
echo "сообщение пользователю" > /dev/tty
$ ./to_screen1 > out 2> err
сообщение пользователю
$ cat out
сообщение на стандартный вывод
$ cat err
сообщение на стандартную ошибку
```

Следующая команда перенаправляет вывод из сценария на экран пользователя:

```
exec > /dev/tty
```

Помещение этой команды в начало предыдущего сценария изменяет маршрут вывода. В сценарии **to_screen2** команда **exec** перенаправляет стандартный вывод на

экран пользователя, поэтому фрагмент `> /dev/tty` становится лишним. Весь вывод, следующий за командой exec, отправляемый на стандартный вывод, попадает на /dev/tty (на экран). Ввод на стандартную ошибку при этом не затрагивается.

```
$ cat to_screen2
exec > /dev/tty
echo "сообщение на стандартный вывод"
echo "сообщение на стандартную ошибку" 1>&2
echo "сообщение пользователю" > /dev/tty

$ ./to_screen2 > out 2> err
сообщение на стандартный вывод
сообщение пользователю
```

Один из недостатков использования команды exec для перенаправления вывода на /dev/tty заключается в том, что весь последующий вывод перенаправляется до тех пор, пока команда exec не будет использована в сценарии еще раз.

Можно также перенаправить ввод, с которого производится чтение (стандартный ввод), чтобы он поступал с /dev/tty (с клавиатуры):

```
read name < /dev/tty
```

или

```
exec < /dev/tty
```

trap: Перехват сигнала

Сигнал — это сообщение процессу о происходящем. Linux использует сигналы для сообщения о прерываниях, созданных пользователем (например, при нажатии клавиши прерывания), а также о неудавшихся системных вызовах, нарушенных каналах, недопустимых инструкциях и других обстоятельствах. Встроенная команда trap (в tcsh используется onintr) отлавливает (перехватывает) один или несколько сигналов, позволяя направлять действия, предпринимаемые сценарием при получении указанного сигнала.

В этом разделе рассматриваются шесть сигналов, имеющих значение при работе со сценариями оболочки. Эти сигналы, номера, которые система им часто приписывает, и условия, при которых обычно генерируется каждый сигнал, перечислены в табл. 10.5. Чтобы вывести список всех имен сигналов, нужно ввести команду `kill -l` (буква «l» в нижнем регистре»), `trap -l` (буква «l» в нижнем регистре») или `man 7 signal`.

Таблица 10.5. Сигналы

Тип	Имя	Номер	Условия генерации
Ненастоящий сигнал	EXIT	0	Выход, вызванный командой exit или достижением конца программы (ненастоящий, но полезный сигнал для перехвата)
Разъединение (Hang up)	SIGHUP или HUP	1	Разъединение на линии
Прерывание терминала	SIGINT или INT	2	Нажатие клавиши прерывания (обычно CONTROL+C)
Выход (Quit)	SIGQUIT или QUIT	3	Нажатие клавиши выхода (обычно CONTROL+SHIFT+| или CONTROL+SHIFT+\)

Тип	Имя	Номер	Условия генерации
Уничтожение (Kill)	SIGKILL или KILL	9	Встроенная команда kill с ключом -9 (не может быть перехвачена; используется только в крайнем случае)
Программное завершение (Software termination)	SIGTERM или TERM	15	Стандартная настройка команды kill
Остановка (Stop)	SIGTSTP или TSTP	20	Нажатие клавиши приостановки (обычно CONTROL+Z)
Отладка (Debug)	DEBUG		Выполнение *команд*, указанных в инструкции trap после каждой команды (это не сигнал, но применяется в команде trap)
Ошибка (Error)	ERR		Выполнение *команд*, указанных в инструкции trap после каждой команды, возвращающей ненулевой статус выхода (это не сигнал, но применяется в команде trap)

При перехвате сигнала сценарий осуществляет то действие, которое вами указано: он может удалить файлы или завершить, если нужно, какую-нибудь другую обработку данных, вывести сообщение, немедленно прекратить выполнение или игнорировать сигнал. Если команда trap в сценарии не применяется, то любой из шести настоящих сигналов, перечисленных в табл. 10.5 (то есть не EXIT, DEBUG или ERR), завершит выполнение сценария. Поскольку процесс не может перехватить сигнал KILL, в качестве крайнего средства для завершения выполнения сценария или другого процесса можно воспользоваться командой `kill -KILL` (или `kill -9`). (См. раздел «kill: Прерывание процесса» на с. 475.)

Для команды trap используется следующий синтаксис:

trap ['команды'][сигнал]

Необязательный аргумент *команды* определяет те команды, которые выполняются оболочкой при перехвате одного из сигналов, указанных в аргументе *сигнал*. В качестве *сигнала* может указываться имя или номер сигнала, например INT или 2. Если *команды* отсутствуют, trap переключает перехват на исходные условия, которые обычно заключаются в выходе из сценария.

Кавычки Встроенная команда trap не требует наличия одинарных кавычек вокруг *команд*, как показано в предыдущем примере синтаксиса, но их использование считается нормальной практикой. Одинарные кавычки заставляют расширять переменные оболочки при поступлении сигнала, а не при вычислении аргументов для команды trap. Даже если в *командах* не используются переменные оболочки, нужно заключать в одинарные или двойные кавычки любую команду, воспринимающую аргументы. Заключенные в кавычки команды заставляют оболочку передавать команде trap всю команду как единый аргумент.

После выполнения *команд* оболочка продолжает выполнение сценария с того места, где оно остановилось. Если нужно, чтобы команда trap предотвратила выход из сценария при получении сигнала, но при этом не запускала никаких *команд*, можно указать ей нулевую (пустую) строку команд, как показано в сценарии locktty (см. раздел «Управляющая структура until» на с. 436). Следующая команда перехватывает сигнал номер 15, после чего продолжается выполнение сценария:

```
trap '' 15
```

В следующем сценарии показано, как встроенная команда trap может перехватить сигнал прерывания терминала (2). Для указания этого сигнала можно использовать SIGINT, INT или 2. Этот сценарий возвращает статус выхода 1:

```
$ cat inter
#!/bin/bash
trap 'echo ПРОГРАММА ПРЕРВАНА; exit 1' INT
while true
do
    echo "Программа работает."
    sleep 1
done
$ ./inter
Программа работает.
Программа работает.
Программа работает.
CONTROL+C
ПРОГРАММА ПРЕРВАНА
$
```

Встроенная команда : (нулевая строка)

Во второй строке сценария inter команда trap настраивается на сигнал прерывания терминала, и при этом используется INT. Когда команда trap перехватывает сигнал, оболочка выполняет две команды, находящиеся в ней между одинарными кавычками. Встроенная команда echo выводит сообщение ПРОГРАММА ПРЕРВАНА, а команда exit завершает работу оболочки, выполняющей сценарий, и родительская оболочка выводит приглашение. Если бы здесь отсутствовала команда exit, оболочка после вывода сообщения вернула бы управление циклу while. Этот цикл повторялся бы до тех пор, пока сценарий не получил бы сигнал, поскольку утилита true всегда возвращает статус выхода *true*. Вместо true можно использовать встроенную команду : (нулевая строка), которая записывается как двоеточие и всегда возвращает статус 0 (*true*).

Встроенная команда trap при преждевременном завершении сценария часто удаляет временные файлы, обеспечивая таким образом чистоту файловой системы. В следующем сценарии оболочки по имени addbanner для удаления временных файлов при нормальном завершении сценария и при сигналах разъединения, прерывания терминала, выхода или программного завершения, используются две команды trap:

```
$ cat addbanner
#!/bin/bash
script=$(basename $0)

if [ ! -r "$HOME/banner" ]
    then
        echo "$script: нужны права на чтение файла $HOME/banner" 1>&2
        exit 1
fi

trap 'exit 1' 1 2 3 15
trap 'rm /tmp/$$.$script 2> /dev/null' 0

for file
```

```
do
    if [ -r "$file" -a -w "$file" ]
        then
            cat $HOME/banner $file > /tmp/$$.$script
            cp /tmp/$$.$script $file
            echo "$script: заголовок добавлен к $file" 1>&2
        else
            echo "$script: нужны права на чтение и запись файла $file" 1>&2
        fi
done
```

При вызове с одним и более аргументов в виде имен файлов сценарий addbanner осуществляет циклический перебор файлов, добавляя в начало каждого из них заголовок. Этот сценарий полезен при использовании стандартного формата в начале ваших документов, например стандартного оформления деловых записок, или при желании добавить стандартный заголовок к сценариям оболочки. Заголовок хранится в файле по имени ~/banner. Поскольку в сценарии addbanner используется переменная HOME, в которой содержится полное имя личного каталога пользователя, сценарий может использоваться без модификации сразу несколькими пользователями. Если у пользователя Max был бы написан сценарий, где вместо $HOME использовалась бы строка /home/max, а затем этот сценарий был бы передан пользователю Zach, то либо Zach должен был бы изменить этот сценарий, либо сценарию addbanner пришлось бы использовать файл заголовка пользователя Max при запуске этого сценария пользователем Zach (если предположить, что у пользователя Zach есть права на чтение этого файла).

Первая команда trap в сценарии addbanner заставляет его выйти со статусом 1 при получении сигнала разъединения, сигнала программного прерывания (прерывания терминала или сигнала выхода) или сигнала программного завершения. Вторая команда trap использует в качестве номера сигнала 0, что заставляет ее выполнить команды, указанные ей в качестве аргумента, *при любом* выходе из сценария, при получении им команды exit или при достижении конца сценария. В результате совместной работы эти команды traps удаляют временные файлы как в случае нормального завершения сценария, так и в случае его преждевременного завершения. Стандартная ошибка второй команды trap при ее попытке удаления несуществующего временного файла отправляется на /dev/null. В таком случае команда rm отправляет сообщение об ошибке на стандартную ошибку, но поскольку стандартная ошибка перенаправлена, пользователь этого сообщения не видит. Еще один пример применения команды trap показан в сценарии locktty в разделе «Управляющая структура until».

kill: Прерывание процесса

Встроенная команда kill отправляет сигнал процессу или заданию и использует следующий синтаксис:

*kill [–**сигнал**] **PID***

где *сигнал* — это имя сигнала или его номер (например, INT или 2), а *PID* — это идентификационный номер процесса, получающего сигнал. Вместо *PID* можно указать в виде *%n* номер задания (см. раздел «Номер задания, PID-номер» на с. 155). Если

опустить *сигнал*, команда kill отправляет сигнал TERM (программное завершение, номер 15). Дополнительные сведения об именах сигналов и их номерах даны в табл. 10.5. Следующая команда отправляет сигнал TERM заданию номер 1, независимо от того, в каком режиме оно выполняется, в активном (запущено) или в фоновом (запущено или остановлено):

```
$ kill -TERM %1
```

Поскольку TERM является для команды kill сигналом по умолчанию, эту команду можно также ввести в виде `kill %1`. Для вывода списка сигналов нужно ввести команду `kill -l` (буква «l» в нижнем регистре).

Прерываемая программа может оставить все в непредсказуемом состоянии: могут быть оставлены временные файлы (обычно они удаляются), и могут быть изменены права доступа. Грамотно написанные приложения перехватывают или обнаруживают сигналы и наводят перед выходом полный порядок. Наиболее аккуратно написанные приложения перехватывают сигналы INT, QUIT и TERM.

Чтобы завершить программу, сначала нужно попытаться послать сигнал INT (нажать комбинацию CONTROL+C, если задание запущено в фоновом режиме). Поскольку приложение может быть написано с учетом игнорирования подобных сигналов, может пригодиться сигнал KILL, который не может быть перехвачен или проигнорирован; он является «гарантированным прерыванием». Дополнительные сведения о команде kill даны в части V. Там же приводятся сведения и о команде killall.

getopts: Разбор ключей

Встроенная команда getopts (в tcsh такой команды нет) проводит разбор аргументов командной строки, упрощая написание программ, которые придерживаются соглашения по аргументам, принятого для операционной системы Linux. Для команды getopts используется следующий синтаксис:

getopts ***строка_ключей имя_переменной [аргумент ...]***

где *строка_ключей* — это список приемлемых букв ключей, *имя_переменной* — это указание переменной, принимающей ключи по одному, и *аргумент* — это дополнительный список обрабатываемых параметров. Если *аргумент* отсутствует, команда getopts обрабатывает аргументы командной строки. Если *строка_ключей* начинается с двоеточия (:), то сценарий должен заняться генерацией сообщений об ошибках, в противном случае генерацией этих сообщений займется команда getopts.

Для отслеживания и хранения значений, имеющих отношения к ключам, встроенная команда getopts использует переменные OPTIND (индекс ключа) и OPTARG (аргумент ключа). При запуске сценария оболочки значение переменной OPTIND равно 1. Когда в очередной раз вызывается команда getopts, и она определяет местонахождение аргумента, значение переменной OPTIND увеличивается на единицу, чтобы индекс указывал на следующий ключ, подлежащий обработке. Если ключ принимает аргумент, оболочка bash присваивает значение аргумента переменной OPTARG.

Чтобы показать, что ключ принимает аргумент, нужно за соответствующей буквой в *строке_ключей* поставить двоеточие (:). *Строка_ключей* `dxo:lt:r` показывает, что команда getopts должна искать ключи -d, -x, -o, -l, -t и -r, а также, что ключи -o и -t принимают аргументы.

Использование команды getopts в качестве *тест-команды* в управляющей структуре while позволяет осуществлять последовательный перебор ключей. Встроенная команда getopts проверяет список ключей на наличие ключей, присутствующих в *строке_ключей*. При каждом проходе цикла команда getopts сохраняет букву найденного ключа в переменной *имя_переменной*.

Предположим, что нужно написать программу, которая может использовать три ключа:

1. Ключ -b, показывающий, что программа должна проигнорировать пробельные символы в начале строк ввода.
2. Ключ -t, за которым следует имя каталога, показывающий, что программа должна сохранять временные файлы в этом каталоге. В противном случае она должна использовать каталог /tmp.
3. Ключ -u, показывающий, что программа должна переводить весь вывод в верхний регистр.

Кроме этого, программа должна игнорировать все остальные ключи и обрабатывать конец ключей при встрече двух дефисов (--).

Задача состоит в написании фрагмента программы, который определяет, какие ключи были предоставлены пользователем. В следующем решении команда getopts не используется:

```
SKIPBLANKS=
TMPDIR=/tmp
CASE=lower
while [[ "$1" = -* ]]     # [[ = ]] проверка соответствия шаблону
do
    case $1 in
        -b) SKIPBLANKS=TRUE ;;
        -t) if [ -d "$2" ]
            then
                TMPDIR=$2
                shift
            else
                echo "$0: -t принимает в качестве аргумента каталог." >&2
                exit 1
            fi ;;
        -u) CASE=upper ;;
        --) break   ;;      # Прекращение обработки ключей
         *) echo "$0: Неправильный ключ $1 проигнорирован." >&2 ;;
        esac
    shift
done
```

В этом фрагменте программы используется цикл для проверки и сдвига аргументов до тех пор, пока у аргумента не встретится значение --. Пока аргумент не будет состоять из двух дефисов, программа продолжает выполнять инструкцию case, с помощью которой проверяется наличие возможных ключей. При соответствии шаблону -- происходит выход из цикла while. С помощью шаблона * определяется любой ключ; он стоит на самом последнем месте для отлова любых неизвестных ключей, вывода сообщения об ошибке и допуска к продолжению процесса. При каждом проходе цикла

для доступа к следующему аргументу программа использует команду смещения shift. Если ключ принимает аргумент, программа использует дополнительное смещение, чтобы добраться до этого аргумента.

Следующая программа обрабатывает такие же ключи, но использует для этого команду getopts:

```
SKIPBLANKS=
TMPDIR=/tmp
CASE=lower
while getopts :bt:u arg
do
    case $arg in
        b)     SKIPBLANKS=TRUE ;;
        t)     if [ -d "$OPTARG" ]
               then
               TMPDIR=$OPTARG
           else
               echo "$0: $OPTARG не является каталогом." >&2
               exit 1
           fi ;;
        u)     CASE=upper ;;
        :)     echo "$0: ключу -$OPTARG нужно предоставить аргумент." >&2
           exit 1 ;;
        \?) echo "Неправильный ключ -$OPTARG проигнорирован." >&2 ;;
        esac
done
```

В этой версии программы структура while производит вычисление встроенной команды getopts при каждой передаче управления в начало цикла. Встроенная команда getopts использует переменную OPTIND для отслеживания индекса аргумента, обрабатываемого при ее следующем вызове. В этом примере вызывать команду shift не нужно.

В версии сценария с использованием команды getopts шаблоны структуры case не начинаются с дефиса, потому что значение переменной arg содержит просто букву ключа (команда getopts удаляет дефис). Кроме этого, команда getopts распознает -- как конец ключей, поэтому указывать эту комбинацию явным образом, как это делалось в инструкции case в первом примере, уже не нужно.

Поскольку команде getopts сообщено, какие ключи разрешены и какие из них требуют использования аргументов, она может обнаружить ошибки в командной строке и обработать их двумя способами. В этом примере для определения, что именно проверять и как обрабатывать ошибки, в *строке_ключей* используется лидирующее двоеточие. Когда команда getopts обнаруживает неправильный ключ, она устанавливает значение переменной *имя_переменной* в ?, а переменной OPTARG присваивает значение буквы ключа. При обнаружении ключа с отсутствующим аргументом команда getopts устанавливает значение переменной *имя_переменной* в :, а переменной OPTARG присваивает значение ключа, которому не хватает аргумента.

Шаблон \? определяет действие, предпринимаемое при обнаружении командой getopts неправильного ключа. Шаблон : определяет действие, предпринимаемое командой getopts при обнаружении отсутствия аргумента ключа. В обоих случаях команда getopts не выдает никаких сообщений об ошибках, возлагая эту задачу на вас.

Если опустить в *строке_ключей* лидирующее двоеточие, то и неправильный ключ, и отсутствующий аргумент ключа приведут к присваиванию переменной *имя_переменной* значения ?. Значение переменной OPTARG не будет установлено, и команда getopts запишет в стандартную ошибку свое собственное диагностическое сообщение. Обычно такой метод менее предпочтителен, поскольку ситуация с тем, что увидит пользователь при возникновении ошибки, становится менее контролируемой.

Использование команды getopts не обязательно должно привести к сокращению объема программы. Ее принципиальное преимущество заключается в предоставлении унифицированного программного интерфейса и во внедрении стандартной обработки ключей.

Список избранных встроенных команд

В табл. 10.6 перечислены некоторые встроенные команды оболочки bash. Чтобы определить, что команда запускает встроенную команду, используется команда type (см. раздел «type: Вывод информации о команде»). Инструкции по выводу полного списка встроенных команд даны в главе 5, в разделе «Вывод списка команд, встроенных в оболочку bash».

Таблица 10.6. Встроенные команды оболочки bash

Встроенная команда	Функция
:	Возвращение 0 или *true* (см. раздел «Встроенная команда : (нулевая строка)» на с. 464)
. (точка)	Выполнение сценария оболочки как части текущего процесса (см. раздел «Утилита . (точка) или source: Запуск файла запуска в текущей оболочке» на с. 287)
bg	Перевод приостановленного задания в фоновый режим выполнения (см. раздел «bg: Отправка задания в фоновый режим» на с. 301)
break	Осуществление выхода из циклической структуры управления (см. раздел «Инструкции break и continue» на с. 438)
cd	Смена рабочего каталога (см. раздел «cd: Смена рабочего каталога» на с. 108)
continue	Начало следующей итерации циклической структуры управления (см. раздел «Инструкции break и continue» на с. 438)
echo	Вывод собственных аргументов (см. раздел «echo: Отображение текста» на с. 77)
eval	Сканирование и вычисление командной строки (см. раздел «Встроенная команда eval» на с. 346)
exec	Выполнение сценария оболочки или программы вместо текущего процесса (см. раздел «exec: Выполнение команды или перенаправление описателей файлов» на с. 469)
exit	Выход из текущей оболочки (обычно то же самое, что и при нажатии комбинации CONTROL+D из интерактивной оболочки; см. раздел «$?: Статус выхода» на с. 469)
export	Помещение значения переменной в пространство вызова (превращение переменной в глобальную переменную; см. раздел «Локальность переменных» на с. 454)
fg	Перевод задания из фонового в активный режим (см. раздел «fg: Перевод задания в активный режим» на с. 300)
getopts	Разбор аргументов, переданных сценарию оболочки (см. раздел «getopts: Разбор ключей» на с. 476)
jobs	Вывод списка фоновых заданий (см. раздел «Управление заданиями» на с. 299)

продолжение

Таблица 10.6 (*продолжение*)

Встроенная команда	Функция
kill	Отправка сигнала процессу или заданию (см. часть V)
pwd	Вывод имени рабочего каталога (см. раздел «Рабочий каталог» на с. 102)
read	Чтение строки из стандартного ввода (см. раздел «read: Получение пользовательского ввода»)
readonly	Объявление переменной только для чтения (см. раздел «Атрибуты переменных» на с. 309)
set	Установка флажков оболочки или переменных, являющихся аргументами командной строки; без аргументов — вывод списка всех переменных (см. раздел «set ±o: Включение и выключение свойств оболочки» на с. 348, раздел «Подстановка переменных» на с. 384 и раздел данной главы «set: Инициализация аргументов командной строки» на с. 461
shift	Смещение каждого аргумента командной строки (см. раздел «shift: Смещение аргументов командной строки» на с. 460)
test	Сравнение аргументов (см. раздел «Встроенная команда test» на с. 432, а также часть V)
times	Вывод суммарного времени работы текущей оболочки и ее дочерних оболочек
trap	Перехват сигнала (см. раздел «trap: Перехват сигнала» на с. 472)
type	Вывод информации о том, как каждый аргумент будет интерпретироваться в качестве команды (см. раздел «type: Вывод информации о команде» на с. 465)
umask	Возвращение значения маски прав доступа, используемой при создании файла (см. часть V)
unset	Удаление переменной или функции (см. раздел «unset: Удаление переменной» на с. 308
wait	Ожидание завершения фонового процесса (см. табл. 9.10 на с. 407)

Выражения

Выражения включают в себя константы, переменные и операторы, которые оболочка в состоянии обработать и вернуть значение. В этом разделе рассматриваются арифметические, логические и условные выражения, а также операторы оболочки bash, которые перечислены в табл. 10.8.

Арифметические вычисления

Оболочка Bourne Again Shell может выполнять арифметические присваивания и производить вычисления многих арифметических выражений различного типа, при этом во всех этих операциях используются целые числа. Оболочка выполняет арифметические присваивания несколькими способами. Один из них предусматривает использование аргументов для встроенной команды let:

```
$ let «VALUE=VALUE * 10 + NEW"
```

В предыдущем примере переменные VALUE и NEW содержат целочисленные значения. Внутри инструкции let использовать знак доллара ($) перед именами переменных не требуется. Отдельный аргумент или выражение, содержащие пробелы, должны быть

взяты в двойные кавычки. Поскольку пробелы содержатся в большинстве выражений и эти выражения должны быть заключены в двойные кавычки, оболочка bash воспринимает команду ((*выражение*)) в качестве синонима команды let "*выражение*", позволяя избежать необходимости использования и кавычек, и знака доллара:

```
$ ((VALUE=VALUE * 10 + NEW))
```

Везде, где разрешено использование этой команды, можно использовать любую форму и можно, если угодно, удалить пробелы. В следующем примере знак звездочки (*) не нуждается в заключении в кавычки, поскольку оболочка не выполняет расширения имени в правой части присваивания (см. главу 8, раздел «Расширение имени в присваиваниях»):

```
$ let VALUE=VALUE*10+NEW
```

Поскольку каждый аргумент команды let вычисляется как отдельное выражение, в одной строке можно присваивать значения более чем одной переменной:

```
$ let "COUNT = COUNT + 1" VALUE=VALUE*10+NEW
```

Для разделения нескольких присваиваний внутри набора из двойных круглых кавычек следует использовать запятые:

```
$ ((COUNT = COUNT + 1, VALUE=VALUE*10+NEW))
```

СРАВНЕНИЕ АРИФМЕТИЧЕСКОГО ВЫЧИСЛЕНИЯ И АРИФМЕТИЧЕСКОГО РАСШИРЕНИЯ

Арифметическое вычисление отличается от арифметического расширения. Согласно объяснениям, которые даны в главе 8, в разделе «Арифметическое расширение» на с. 356, при арифметическом расширении используется синтаксис $((выражение)), вычисляется выражение и $((выражение)) заменяется результатом. Арифметическое расширение можно использовать для вывода значения выражения или для присваивания этого значения переменной.

Арифметическое вычисление использует синтаксис let выражение или ((выражение)), вычисляет выражение и возвращает код статуса. Арифметическое вычисление можно использовать для выполнения логических сравнений или присваиваний.

Логические выражения

Синтаксис ((*выражение*)) можно использовать для логических выражений, хотя эта задача зачастую возлагается на синтаксис [[*выражение*]]. В следующем примере приводится расширенная версия сценария age_check (см. раздел «Арифметическое расширение» на с. 356), включающая в дополнение к арифметическому расширению еще и арифметическое вычисление:

```
$ cat age2
#!/bin/bash
echo -n "How old are you? "
read age
if ((30 < age && age < 60)); then
       echo "Wow, in $((60-age)) years, you'll be 60!"
   else
       echo "You are too young or too old to play."
fi

$ ./age2
How old are you? 25                    Сколько вам лет?
You are too young or too old to play. Вы слишком молоды или стары, чтобы играть
```

Тест-команда для структуры if вычисляет два логических сравнения, объединенных булевым И и возвращающих 0 (*true*), если они оба вычисляются в *true*, или 1 (*false*) в противном случае.

Логические вычисления (условные выражения)

Для условного выражения используется следующий синтаксис:

```
[[ выражение ]]
```

где *выражение* — это булево (логическое) выражение. Внутри выражения впереди имен переменных нужно ставить знак доллара ($). Результат выполнения этой встроенной команды, как и встроенной команды test, содержится в статусе возврата. Условия, разрешенные внутри квадратных скобок, практически составляют поднабор тех условий, которые разрешены во встроенной команде test (см. часть V). Там где встроенная команда использует в качестве булева оператора И (AND) ключ -a, команда [[*выражение*]] использует символы &&. Точно так же, там где команда test использует ключ -o в качестве булева оператора ИЛИ (OR), команда [[*выражение*]] использует символы ||.

Чтобы посмотреть, как работает условное выражение, заменим строку, проверяющую возраст в сценарии age2, следующим условным выражением. Лексемы [[и]] нужно окружить пробелами или символом завершения команды и поставить знак доллара перед именем каждой переменной:

```
if [[ 30 < $age && $age < 60 ]]; then
```

Можно также воспользоваться операторами отношений команды test -gt, -ge, -lt, -le, -eq и -ne:

```
if [[ 30 -lt $age && $age -lt 60 ]]; then
```

Сравнение строк

Встроенная команда test проверяет строки на равенство. Синтаксис [[*выражение*]] добавляет сравнительные тесты для строковых операторов. Операторы > и < сравнивают строки в алфавитном порядке (например, "aa" < "bbb"). Оператор = проверяет соответствие шаблону, а не просто равенство: [[*строка* = *шаблон*]], и вычисляется в *true*, если *строка* соответствует *шаблону*. Этот оператор на обладает симметрией, поэтому *шаблон* должен быть справа от знака равенства. Например, [[artist = a*]] вычисляется в *true* (= 0), а [[a* = artist]] вычисляется в *false* (= 1):

```
$ [[ artist = a* ]]
$ echo $?
0
$ [[ a* = artist ]]
$ echo $?
1
```

В следующем примере используется список команд, который начинается с объединенного условия. Это условие проверяет существование каталога bin и файла src/myscript.bash. Если в результате проверки возвращается *true*, команда cp копирует файл src/myscript.bash в файл bin/myscript. Если копирование будет успешным, команда chmod превращает файл myscript в исполняемый. Если какой-нибудь из этих этапов проидет неудачно, команда echo выведет сообщение.

```
$ [[ -d bin && -f src/myscript.bash ]] && cp src/myscript.bash \
bin/myscript && chmod +x bin/myscript || echo "Создать исполняемую \
версию myscript невозможно"
```

Соответствие строки шаблону

Оболочка Bourne Again Shell предоставляет операторы соответствия строк шаблонам, которые могут работать с именами и другими строками. Эти операторы могут удалять из строки префиксы или суффиксы, которые соответствуют шаблонам. В табл. 10.7 перечислены четыре оператора.

Таблица 10.7. Строковые операторы

Оператор	Функция
#	Удаление минимально соответствующих префиксов
##	Удаление максимально соответствующих префиксов
%	Удаление минимально соответствующих суффиксов
%%	Удаление максимально соответствующих суффиксов

Для этих операторов используется следующий синтаксис:

${имя_переменной оператор шаблон}

где *оператор* — один из операторов из табл. 10.7, а *шаблон* — шаблон соответствия, подобный тому, что используется для генерации имен файлов. Эти операторы обычно используются для работы с именами, для извлечения или удаления компонентов или для изменения суффиксов:

```
$ SOURCEFILE=/usr/local/src/prog.c
$ echo ${SOURCEFILE#/*/}
local/src/prog.c
$ echo ${SOURCEFILE##/*/}
prog.c
$ echo ${SOURCEFILE%/*}
/usr/local/src
$ echo ${SOURCEFILE%%/*}

$ echo ${SOURCEFILE%.c}
/usr/local/src/prog
$ CHOPFIRST=${SOURCEFILE#/*/}
$ echo $CHOPFIRST
local/src/prog.c
$ NEXT=${CHOPFIRST%%/*}
$ echo $NEXT
local
```

А здесь оператор длины строки ${#*имя*} заменяется количеством символов в значении переменной *имя*:

```
$ echo $SOURCEFILE
/usr/local/src/prog.c
$ echo ${#SOURCEFILE}
21
```

Операторы

Арифметическое расширение и арифметическое вычисление в bash используют точно такой же синтаксис, приоритетность операторов и ассоциативность выражений, что и вычисления в языке C. В табл. 10.8 перечислены операторы в порядке убывания приоритетности вычислений. Операторы в каждой группе имеют одинаковую приоритетность. Чтобы изменить порядок вычислений, можно внутри выражения использовать круглые скобки.

Таблица 10.8. Операторы

Тип оператора — оператор	Функция
Пост	
var++	Постинкремент
var--	Постдекремент
Пре	
++*var*	Преинкремент
--*var*	Предекремент
Унарный	
-	Унарный минус
+	Унарный плюс
Отрицание	
!	Булево НЕ (логическое отрицание)
~	Дополнение (поразрядное отрицание)
Возведение в степень	
**	Возведение в степень
Умножение, деление, получения остатка от деления	
*	Умножение
/	Деление
%	Получение остатка от деления
Сложение, вычитание	
-	Вычитание
+	Сложение
Поразрядный сдвиг	
<<	Поразрядный сдвиг влево
>>	Поразрядный сдвиг вправо
Сравнение	
<=	Меньше чем или равно
>=	Больше чем или равно
<	Меньше чем
>	Больше чем
==	Равенство
!=	Неравенство

Тип оператора — оператор	Функция
Поразрядный	
&	Поразрядное И (AND)
^	Поразрядное исключающее ИЛИ (XOR)
|	Поразрядное ИЛИ (OR)
Булев (логический)	
&&	Булево И (AND)
||	Булево ИЛИ (OR)
Условного вычисления	
? :	Трехкомпонентный оператор
Присваивания	
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=	Присваивание
Запятая	
,	Запятая

Канал

Лексема канала имеет более высокий приоритет, чем операторы. Каналы можно использовать в команде повсюду, применяя при этом простые команды. Например, командная строка

```
$ cmd1 | cmd2 || cmd3 | cmd4 && cmd5 | cmd6
```

интерпретируется так, как будто была набрана строка

```
$ ((cmd1 | cmd2) || (cmd3 | cmd4)) && (cmd5 | cmd6)
```

НЕ ПОЛАГАЙТЕСЬ НА ПРАВИЛА ПРИОРИТЕТНОСТИ: ИСПОЛЬЗУЙТЕ КРУГЛЫЕ СКОБКИ

При использовании сложных команд не стоит полагаться на правила приоритетности. Лучше воспользоваться круглыми скобками, чтобы явным образом установить тот порядок, в котором нужно заставить оболочку интерпретировать команды.

Инкремент и декремент

Операторы постинкремента, постдекремента, преинкремента и предекремента работают с переменными. Пре- операторы, которые ставятся перед именем переменной (как в примерах ++COUNT и --VALUE), сначала изменяют значение переменной (++ прибавляет 1, а -- вычитает 1), а затем предоставляют результат для использования его в выражении. Пост- операторы ставятся после имени переменной (как в примерах COUNT++ и VALUE--); они сначала предоставляют неизмененное значение переменной для использования в выражении, а затем изменяют значение переменной.

```
$ N=10
$ echo $N
10
$ echo $((--N+3))
12
$ echo $N
9
```

```
$ echo $((N++ - 3))
6
$ echo $N
10
```

Остаток от деления Оператор остатка от деления (%) выдает остаток от деления его первого операнда на второй. Например, выражение `$((15%7))` имеет значение 1.

Булевы операции Результат булевой операции либо 0 (*true*), либо 1 (*false*). Булевы операторы && (И) и || (ИЛИ) называются *закороченными*. Если результат использования одного из этих операторов может быть понятен при рассмотрении только одного левого операнда, правый операнд не вычисляется. Оператор && заставляет оболочку тестировать статус выхода той команды, которая ему предшествует. Если команда завершила свою работу успешно, bash выполняет следующую команду, в противном случае она пропускает оставшиеся в командной строке команды. Эту конструкцию можно использовать для условного выполнения команд.

```
$ mkdir bkup && cp -r src bkup
```

Эта команда создает каталог bkup. Если команда mkdir отработает успешно, содержимое каталога src будет в рекурсивном режиме скопировано в каталог bkup.

Разделитель || также заставляет bash протестировать статус выхода из первой команды, но производит обратный эффект: оставшаяся команда (команды) выполняется только в том случае, если первая команда завершится неудачно (то есть статус выхода из нее будет ненулевым).

```
$ mkdir bkup || echo "Отказ mkdir или bkup" >> /tmp/log
```

Статусом выхода из списка команд является статус выхода из последней команды списка. Списки можно группировать с помощью круглых скобок. Для наглядности можно объединить предыдущие два примера:

```
$ (mkdir bkup && cp -r src bkup) || echo "Отказ mkdir" >> /tmp/log
```

В отсутствие круглых скобок, операторы && и || имеют равную приоритетность и группируются слева направо. В следующих примерах используются утилиты true и false. Эти утилиты ничего не делают и возвращают статусы выхода *true* (0) и *false* (1) соответственно:

```
$ false; echo $?
1
```

Переменная $? сохраняет статус выхода из предыдущей команды (см. раздел «$?: Статус выхода» на с. 458). Следующие две команды выдают статус выхода 1 (*false*):

```
$ true || false && false
$ echo $?
1
$ (true || false) && false
$ echo $?
1
```

Следующие две команды выдают статус выхода 0 (*true*):

```
$ false && false || true
$ echo $?
0
```

```
$ (false && false) || true
$ echo $?
0
```

Поскольку у операторов || и && равная приоритетность, круглые скобки в двух предыдущих парах примеров не изменяют порядок выполнения операций.

Поскольку выражение, находящееся справа от закороченного оператора, может никогда не выполняться, нужно быть внимательным при помещении оператора присваивания в это место. В следующем примере показывается, что может произойти:

```
$ ((N=10,Z=0))
$ echo $((N || ((Z+=1)) ))
1
$ echo $Z
0
```

Поскольку значение переменной N не равно нулю, результат операции || (ИЛИ) равен 1 (*true*), независимо от того, какое значение будет в правой части. Как следствие, выражение ((Z+=1)) никогда не вычисляется, и значение переменной Z не увеличивается.

Трехкомпонентный оператор

Трехкомпонентный оператор ? : , решает, какое из двух выражений должно быть вычислено, основываясь при этом на значении, возвращенном третьим выражением:

выражение1 ? выражение2 : выражение3

Если *выражение1* выдает значение *false* (0), вычисляется *выражение3*; в противном случае вычисляется *выражение2*. Значение всего выражения — это значение *выражения2* или *выражения3*, в зависимости от того, какое из них будет вычислено. Если *выражение1* вычисляется в *true*, *выражение3* не вычисляется. Если *выражение1* вычисляется в *false*, то не вычисляется *выражение2*.

```
$ ((N=10,Z=0,COUNT=1))
$ ((T=N>COUNT?++Z:--Z))
$ echo $T
1
$ echo $Z
1
```

Присваивание

Такие операторы присваивания, как +=, являются сокращенной формой записи. Например, N+=3 — это то же самое, что и ((N=N+3)).

Числа с другими основаниями

В следующих командах используется синтаксис *основание#n* для присваивания значений с основанием 2 (двоичных значений). Сначала переменной v1 присваивается значение 0101 (5 десятичное), а затем переменной v2 присваивается значение 0110 (6 десятичное). Десятичные значения проверяются с помощью утилиты echo.

```
$ ((v1=2#0101))
$ ((v2=2#0110))
$ echo "$v1 и $v2"
5 и 6
```

Затем поразрядный оператор И (&) выбирает биты, установленные как в числе 5 (в двоичной записи 0101), так и в числе 6 (0110). В результате получается число 0100 (4 десятичное).

```
$ echo $(( v1 & v2 ))
4
```

Булевский оператор И (&&) выдает результат 1, если оба его операнда имеют ненулевое значение, и результат 0 в противном случае. Включающий поразрядный оператор ИЛИ (|) выбирает биты, которые установлены либо в числе 0101, либо в числе 0110, что приводит к результату 0111 (7 десятичное).

Булевский оператор ИЛИ (||) выдает результат 1, если оба его операнда имеют ненулевое значение, и результат 0 в противном случае.

```
$ echo $(( v1 && v2 ))
1
$ echo $(( v1 | v2 ))
7
$ echo $(( v1 || v2 ))
1
```

Следующий поразрядный исключающий оператор ИЛИ (^) выбирает биты, которые установлены только в одном, а не в обоих операндах 0101 и 0110, что приводит к результату 0011 (3 десятичное). Булевский оператор НЕ (!) выдает результат 1, если его операнд равен 0, и результат 0 в противном случае. Поскольку восклицательный знак в выражении `$(( ! v1 ))` заключен в двойные круглые скобки, он не нуждается в отключении, предотвращающем его интерпретацию оболочкой в качестве события истории. Операторы сравнения выдают результат 1, если сравнение вычисляется в *true*, и результат 0 в противном случае.

```
$ echo $(( v1 ^ v2 ))
3
$ echo $(( ! v1 ))
0
$ echo $(( v1 < v2 ))
1
$ echo $(( v1 > v2 ))
0
```

Программы оболочки

Оболочка Bourne Again Shell обладает многими свойствами, превращающими ее в хороший язык программирования. Структуры, предоставляемые оболочкой bash, не являются некой случайной подборкой, скорее всего, они были выбраны для предоставления большинства структурных возможностей, имеющихся в других процедурных языках, таких как C или Perl. Процедурный язык предоставляет следующие возможности:

- Объявление переменных и констант, присваивание им значений и работу с ними. Оболочка Bourne Again Shell предоставляет строковые переменные наряду с мощными строковыми операторами и целочисленные переменные наряду с полным набором арифметических операторов.

- Разбиение крупных задач на более мелкие за счет создания подпрограмм. Оболочка Bourne Again Shell позволяет создавать функции и вызывать сценарии из других сценариев. Функции оболочки могут вызываться рекурсивно, то есть функции оболочки Bourne Again Shell могут вызывать сами себя. Потребности в рекурсии могут возникать не так уж и часто, но она может позволить легко решить некоторые, несомненно сложные задачи.
- Условное выполнение инструкций с использованием таких управляющих структур, как if.
- Многократное выполнение инструкций с использованием таких управляющих структур, как while и for.
- Перемещение данных в программу и из программы, взаимодействие как с файлами данных, так и с пользователями.

Языки программирования реализуют эти возможности различными путями, но в их основу положен один и тот же замысел. Когда нужно решить задачу путем написания программы, сначала нужно составить алгоритм, то есть придумать процедуру, которая приведет к ее решению. Как правило, один и тот же алгоритм можно реализовать практически одинаково на разных языках программирования, используя те же самые конструкции.

В главе 8 и в данной главе представлено большое количество свойств оболочки bash, многие из которых применимы как при работе в интерактивном режиме, так и при программировании оболочки. В этом разделе разрабатываются две полноценные программы оболочки, демонстрирующие эффективное сочетание некоторых из этих свойств. Программы представлены в виде задач для решения с предоставлением типовых решений.

Рекурсивный сценарий оболочки

Рекурсивная конструкция относится к таким конструкциям, которые определяются на основе самих себя. По-другому можно сказать, что рекурсивная программа — это одна из тех программ, которая вызывает саму себя. Это понятие может показаться каким-то замкнутым кругом, но оно не должно быть таковым. Чтобы избежать замкнутости, определение рекурсивности должно иметь особый случай, который не приводит к самовызову. Рекурсивные идеи появляются и в повседневной жизни. Например, можно определить предка вашей матери, вашего отца или одного из их предков. Это определение не имеет зацикленности, можно вполне однозначно определить, кто ваши предки: ваша мать или ваш отец, или мать вашей матери или отца, или мать вашего отца или его отец и т. д.

Многие системные утилиты Linux могут работать рекурсивно. К примеру, обратите внимание на ключ `-R` утилит chmod, chown и cp, описание которых дано в части V.

Решите следующую задачу, используя рекурсивную функцию оболочки:

> Напишите функцию оболочки по имени makepath, которая на основе заданного путевого имени создает из всех компонентов этого путевого имени каталоги. Например, команда `makepath a/b/c/d` должна создать каталоги a, a/b, a/b/c и a/b/c/d. (Подобным образом создаются каталоги с помощью команды mkdir с ключом `-p`. Решите эту задачу без использования команды `mkdir -p`.)

Один из алгоритмов решения рекурсивной задачи заключается в следующем:

1. Проанализировать аргумент пути. Если он представлен нулевой строкой или если в нем указан существующий каталог, ничего не делать и вернуть управление.
2. Если аргумент пути представлен простой компонентой пути, создать каталог (используя команду mkdir) и вернуть управление.
3. Во всех остальных случаях вызвать функцию makepath, используя в качестве нового аргумента префикс пути. Этот шаг в конечном итоге создает все каталоги вплоть до последней компоненты, каталог которой можно затем создать, используя команду mkdir.

В общем случае, рекурсивная функция должна вызывать саму себя с простой версией поставленной перед ней задачи до тех пор, пока она в конечном итоге не будет вызвана в простом варианте, не требующем ее самовызова. Следующий далее пример является одним из возможных решений, основанных на этом алгоритме:

makepath

```
# Это функция
# Введите ее с клавиатуры и не запускайте как сценарий оболочки
#
function makepath()
{
    if [[ ${#1} -eq 0 || -d "$1" ]]
        then
            return 0        # Никаких действий
    fi
    if [[ "${1%/*}" = "$1" ]]
        then
            mkdir $1
            return $?
    fi
    makepath ${1%/*} || return 1
    mkdir $1
    return $?
}
```

В тесте на простую компоненту (инструкция if в середине функции) левое выражение является аргументом после отбрасывания самого короткого суффикса, который начинается с символа / (см. табл. 10.7). Если такого символа не существует (например, если параметр $1 равен max), отбрасывать нечего и обе части равны. Если аргумент является простым именем файла, перед которым стоит слэш, как, например, /usr, выражение ${1%/*} вычисляется в нулевую строку. Чтобы функция в данном случае работала, нужно предусмотреть две вещи: поместить левое выражение в кавычки и обеспечить разумное поведение рекурсивной функции при передаче ей в качестве аргумента нулевой строки. Как правило, хорошие программы отличаются устойчивостью: они должны быть готовы к сомнительному, неправильному или бессмысленному вводу и вести себя в подобных случаях соответствующим образом.

Путем ввода следующей команды из оболочки, в которой ведется работа, включается отладочная трассировка, позволяющая отследить рекурсивную работу:

```
$ set -o xtrace
```

(Чтобы выключить режим отладки, нужно дать такую же команду, но заменить дефис знаком плюс (+).) При включении режима отладки оболочка выводит каждую строку в ее расширенной форме и выполняет строку. Каждая строка отладочного вывода предваряется знаком плюс (+).

В следующем примере первая строка, начинающаяся со знака +, показывает оболочку, вызывающую функцию makepath. Эта функция сначала вызывается из командной строки с аргументами a/b/c. Затем она вызывает саму себя с аргументами a/b, и наконец, с аргументом a. Вся работа делается (с использованием команды mkdir) при возвращении из каждого вызова функции makepath.

```
$ ./makepath a/b/c
+ makepath a/b/c
+ [[ 5 -eq 0 ]]
+ [[ -d a/b/c ]]
+ [[ a/b = \a\/\b\/\c ]]
+ makepath a/b
+ [[ 3 -eq 0 ]]
+ [[ -d a/b ]]
+ [[ a = \a\/\b ]]
+ makepath a
+ [[ 1 -eq 0 ]]
+ [[ -d a ]]
+ [[ a = \a ]]
+ mkdir a
+ return 0
+ mkdir a/b
+ return 0
+ mkdir a/b/c
+ return 0
```

Функция опускается вниз по своему рекурсивному пути и снова возвращается наверх.

Будет полезно вызвать makepath с неправильно указанным путем и посмотреть, что из этого получится. В следующем примере, запущенном с включенным отладочным режимом, предпринимается попытка создать путь /a/b. Для создания этого пути нужно создать каталог в корневом каталоге. Пока у вас не будет прав записи в корневой каталог, создание этого каталога будет запрещено.

```
$ ./makepath /a/b
+ makepath /a/b
+ [[ 4 -eq 0 ]]
+ [[ -d /a/b ]]
+ [[ /a = \/\a\/\b ]]
+ makepath /a
+ [[ 2 -eq 0 ]]
+ [[ -d /a ]]
+ [[ '' = \/\a ]]
+ makepath
+ [[ 0 -eq 0 ]]
+ return 0
+ mkdir /a
```

```
mkdir: cannot create directory '/a': Permission denied
                    Невозможно создать каталог '/a': Нет прав доступа
+ return 1
+ return 1
```

Рекурсия останавливается, когда makepath не получает разрешения на создание каталога /a. Возвращенная ошибка проходит весь путь назад, поэтому первоначально вызванная функция makepath осуществляет выход с ненулевым статусом.

ИСПОЛЬЗУЙТЕ В РЕКУРСИВНЫХ ФУНКЦИЯХ ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

В предыдущем примере устранена потенциальная проблема рекурсивной функции, с которой можно столкнуться при ее использовании. При выполнении рекурсивной функции могут быть одновременно активны многие отдельные экземпляры этой функции. Все, кроме одного из них, ждут завершения своих дочерних вызовов.

Поскольку функции запускаются в той же среде окружения, в которой работает вызвавшая их оболочка, то само собой разумеется, что переменные совместно используются как оболочкой, так и вызванной ей функцией. Вследствие этого все экземпляры функций совместно используют одну и ту же копию каждой переменной. Совместное использование переменных может вызвать побочные и, скорее всего, нежелательные эффекты. Как правило, для превращения всех переменных рекурсивной функции в локальные, нужно использовать команду typeset. Дополнительные сведения даны в разделе «Функции» на с. 455.

Сценарий оболочки quiz

Решите следующую задачу, используя сценарий оболочки bash:

> Напишите универсальную программу quiz (контрольный опрос), предоставляющую возможность выбора из нескольких вариантов. Программа должна получать свои вопросы из файлов данных, задавать их пользователю и отслеживать количество правильных и неправильных ответов. Пользователь должен в любое время иметь возможность выхода из программы и получения итоговых результатов на момент выхода.

Конкретная конструкция данной программы и даже подробное описание задачи зависит от количества предлагаемых вариантов: Как программа сможет узнать, какие предметы доступны для контрольного опроса? Как пользователь сможет выбрать предмет? Как программа узнает о завершении опроса? Должна ли программа задавать одни и те же вопросы (по заданному предмету) и каждый раз в том же самом порядке, или она должна их перемешивать?

Разумеется, существует множество приемлемых вариантов реализации поставленной задачи. Следующие подробности конкретизируют спецификацию:

- Каждый предмет будет соотноситься с подкаталогом главного каталога программы quiz. Каталог будет указан в переменной среды QUIZDIR, чье значение по умолчанию будет ~/quiz. Например, могут быть следующие каталоги, соотносящиеся с предметами машиностроение (engineering), искусство (art) и политика (politics): ~/quiz/engineering, ~/quiz/art и ~/quiz/politics. Если нужно, чтобы доступ к каталогу quiz имели все пользователи, его следует поместить в каталог /usr/games (для чего необходимо иметь root-права).
- По каждому предмету должно быть несколько вопросов. Каждый вопрос представлен файлом в каталоге своего предмета.

- В первой строке каждого файла, представляющего вопрос, хранится текст вопроса. Если он занимает более одной строки, разделитель строк нужно отключить с помощью символа обратного слэша. (Такая настройка упростит чтение отдельного вопроса с помощью встроенной команды read.) Вторая строка файла представляет собой целое число, указывающее на количество вариантов ответа. А следующие строки содержат сами варианты ответа. В последней строке содержится правильный ответ. Образец файла вопросов имеет следующий вид:

```
Who discovered the principle of the lever?      Кто открыл принцип рычага?
4
Euclid                                          Эвклид
Archimedes                                      Архимед
Thomas Edison                                   Томас Эдисон
The Lever Brothers                              Компания «Lever Brothers»
Archimedes                                      Архимед
```

- Программа содержит все вопросы в каталоге предмета. Пользователь может прервать работу программы quiz в любой момент, нажав комбинацию CONTROL+C, после чего программа подсчитает итоговый результат, полученный на данный момент, и осуществит выход. Если пользователь не прервет работу программы, она подведет итоги и осуществит выход, когда будут даны ответы на все вопросы по выбранному предмету.
- Перед тем как задать вопросы по предмету, программа их перемешивает.

Основной замысел программы выглядит следующим образом:

1. Инициализация. Предпринимается ряд шагов, таких как установка в нулевые значения счетчиков количества уже заданных вопросов и количества правильных и неправильных ответов. Кроме этого, производится настройка программы на перехват сигнала, генерируемого комбинацией CONTROL+C.
2. Предоставление пользователю возможности выбора предмета и получение пользовательского ответа.
3. Переход к соответствующему каталогу предмета.
4. Определение задаваемых вопросов (то есть имен файлов в этом каталоге). Выстраивание их в произвольном порядке.
5. Многократное предоставление вопросов и запрос ответов до тех пор, пока программа quiz не завершит свою работу или не будет прервана пользователем.
6. Предоставление результатов и выход.

Очевидно, что некоторые шаги (например, шаг 3) реализуются просто, но есть и другие, более сложные шаги (например, шаг 4), которые стоит рассмотреть отдельно. Для каждого сложного шага нужно использовать функции оболочки, а для обработки прерывания, исходящего от пользователя, нужно использовать встроенную команду trap.

Схематически программа, составленная из пустых функций оболочки, имеет следующий вид:

```
function initialize
{
# Инициализация переменных.
}
```

```
function choose_subj
{
# Запись выбранного варианта в стандартный вывод.
}

function scramble
{
# Сохранение перемешанных имен файлов вопросов
# в переменной-массиве по имени questions.
}

function ask
{
# Чтение файла вопроса, задание вопроса и проверка ответа.
# Возвращение 1 при правильном и 0 при неправильном ответе.
# Если встретится испорченный файл вопроса, выход со статусом 2.
}

function summarize
{
# Предоставление оценки, заработанной пользователем.
}

# Основная программа
initialize                          # Шаг 1 основного замысла

subject=$(choose_subj)              # Шаг 2
[[ $? -eq 0 ]] || exit 2            # Если выбор неверен – выход
cd $subject || exit 2               # Шаг 3
echo                                # Пропуск строки
scramble                            # Шаг 4

for ques in ${questions[*]}; do     # Шаг 5
    ask $ques
    result=$?
    (( num_ques=num_ques+1 ))
    if [[ $result == 1 ]]; then
        (( num_correct += 1 ))
    fi
    echo                            # Пропуск строки между вопросами
    sleep ${QUIZDELAY:=1}
done
summarize                           # Шаг 6
exit 0
```

Чтобы облегчить пользователю чтение результатов, в цикл вопросов вставлен вызов команды **sleep**. Он обеспечивает задержку между вопросами на количество секунд, указанное в переменной `$QUIZDELAY` (по умолчанию = 1).

Теперь нужно заполнить отсутствующие части программы. В некотором смысле, эта программа пишется с конца. Сначала в файле появляются элементы конструкции (функции оболочки), и только потом начинается процесс их разработки. Эта практика,

получившая широкое распространение, называется проектированием сверху вниз. При нисходящем проектировании сначала отрабатывается общая схема программы, а конкретные детали предоставляются позже. Таким образом, задача разбивается на более мелкие задачи, каждая из которых может выполняться независимо от всех других. Функции оболочки являются хорошим подспорьем для нисходящего подхода к созданию программы.

Следующий код является одним из вариантов функции инициализации. Команда cd делает каталог, указанный в переменной QUIZDIR, рабочим для всего остального сценария и, если значение QUIZDIR не установлено, использует по умолчанию каталог ~/quiz.

```
function initialize ()
{
trap 'summarize ; exit 0' INT     # Обработка пользовательских прерываний
num_ques=0                        # Количество заданных на данный момент вопросов
num_correct=0                     # Текущее количество правильных ответов
first_time=true                   # true, пока не будет задан первый вопрос
cd ${QUIZDIR:=~/quiz} || exit 2
}
```

Нужно быть готовым к тому, что команда cd выдаст отказ. Возможно, каталог не будет найден или удален другим пользователем. Если команда cd не будет выполнена, то предыдущая функция осуществит выход с кодом статуса 2.

Следующая функция, choose_subj, имеет более сложную конструкцию. Она выводит меню, используя инструкцию select:

```
function choose_subj ()
{
subjects=($(ls))
PS3="Выберите предмет опроса из предыдущего списка: "
select Subject in ${subjects[*]}; do
    if [[ -z "$Subject" ]]; then
        echo "Предмет не выбран. Всего хорошего." >&2
        exit 1
    fi
    echo $Subject
    return 0
done
}
```

Сначала функция использует команду ls и командную подстановку, чтобы поместить список каталогов предметов в массив subjects. Затем структура select (см. раздел «Управляющая структура select» на с. 445) предоставляет пользователю список предметов (имен каталогов, найденных командой ls) и присваивает имя выбранного каталога переменной Subject. И в завершение, функция записывает имя каталога предмета в стандартный вывод. В основной программе для присваивания этого значения переменной subject используется командная подстановка: [subject=$(choose_subj)].

Реализация функции scramble представляет ряд трудностей. В данном решении для хранения имен вопросов в ней используется переменная массива (questions). Функция перемешивает записи массива, используя переменную RANDOM (при каждой

ссылке на RANDOM эта переменная имеет случайное целочисленное значение в диапазоне от 0 до 32767):

```
function scramble ()
{
typeset -i index quescount
questions=($(ls))
quescount=${#questions[*]}        # Количество элементов
((index=quescount-1))
while [[ $index > 0 ]]; do
    ((target=RANDOM % index))
    exchange $target $index
    ((index -= 1))
done
}
```

Эта функция инициализирует переменную массива questions с использованием списка имен файлов (вопросов), находящихся в рабочем каталоге. Переменная quescount получает значение, равное количеству таких файлов. Затем используется следующий алгоритм: значение переменной index постоянно уменьшается на единицу, начиная со значения quescount – 1 (индекса последней записи в переменной массива). Для каждого значения переменной index функция выбирает случайное значение между 0 и index включительно. Команда

```
((target=RANDOM % index))
```

выдает случайное значение между 0 и index – 1 путем использования оператора вычисления остатка (оператора %), получаемого от деления $RANDOM на index. Затем функция производит обмен элементов массива questions, находящихся на позициях target и index. Этот шаг лучше выделить в другую функцию по имени exchange:

```
function exchange ()
{
temp_value=${questions[$1]}
questions[$1]=${questions[$2]}
questions[$2]=$temp_value
}
```

В функции ask также используется структура select. Эта функция читает файл вопроса, указанный в ее аргументе, и использует содержимое этого файла для того, чтобы показать вопрос, получить ответ и определить его правильность (см. код, показанный ниже). Для чтения последовательных строк из файла вопроса, имя которого было передано в качестве аргумента и представлено в функции параметром $1, функция ask использует описатель файла 3. Вопрос считывается в переменную ques, а количество ответов — в переменную num_opts. В функции создается переменная choices, изначально содержащая нулевую строку, к которой последовательно добавляется следующий вариант ответа. Затем переменной PS3 присваивается значение переменной ques, и структура select используется для вывода приглашения пользователю в виде вопроса, являющегося значением переменной ques. Структура select помещает пользовательский ответ в переменную answer, а затем функция проверяет этот ответ на соответствие правильному ответу из файла.

Конструкция переменной choices выбрана с расчетом на устранение потенциальной проблемы. Предположим, что один из ответов содержит пробельные символы, тогда

он может появиться в choices в виде двух и более аргументов. Во избежание этой проблемы, нужно, чтобы переменная choices обязательно была массивом. Всю остальную работу сделает инструкция select:

quiz

```
$ cat quiz
#!/bin/bash
# Чтобы включить отладку, уберите символ # из следующей строки
# set -o xtrace
#==================
function initialize ()
{
trap 'summarize ; exit 0' INT     # Обработка пользовательских прерываний
num_ques=0                        # Количество заданных на данный момент вопросов
num_correct=0                     # Текущее количество правильных ответов
first_time=true                   # true, пока не будет задан первый вопрос
cd ${QUIZDIR:=~/quiz} || exit 2
}
#==================
function choose_subj ()
{
subjects=($(ls))
PS3="Выберите предмет опроса из предыдущего списка: "
select Subject in ${subjects[*]}; do
    if [[ -z "$Subject" ]]; then
        echo "Предмет не выбран. Всего хорошего." >&2
        exit 1
    fi
    echo $Subject
    return 0
done
}
#==================
function exchange ()
{
temp_value=${questions[$1]}
questions[$1]=${questions[$2]}
questions[$2]=$temp_value
}
#==================
function scramble ()
{
typeset -i index quescount
questions=($(ls))
quescount=${#questions[*]}        # Количество элементов
((index=quescount-1))
while [[ $index > 0 ]]; do
    ((target=RANDOM % index))
    exchange $target $index
    ((index -= 1))
done
}
```

```
#=================
function ask ()
{
exec 3<$1
read -u3 ques || exit 2
read -u3 num_opts || exit 2
index=0
choices=()
while (( index < num_opts )) ; do
    read -u3 next_choice || exit 2
    choices=("${choices[@]}" "$next_choice")
    ((index += 1))
done
read -u3 correct_answer || exit 2
exec 3<&-
if [[ $first_time = true ]]; then
    first_time=false
    echo -e "Для выхода можно в любой момент нажать клавишу прерывания.\n"
fi
PS3=$ques"  "              # Превращение $ques в приглашение структуры select
                           # и добавление нескольких пробелов для удобства чтения.
select answer in "${choices[@]}"; do
    if [[ -z "$answer" ]]; then
            echo  Неверный выбор. Пожалуйста, повторите.
        elif [[ "$answer" = "$correct_answer" ]]; then
            echo "Правильно!"
            return 1
        else
            echo "Нет, правилный ответ: $correct_answer."
            return 0
    fi
done
}
#=================
function summarize ()
{
echo                               # Пропуск строки
if (( num_ques == 0 )); then
    echo "Вы не ответили ни на один вопрос"
    exit 0
fi
(( percent=num_correct*100/num_ques ))
echo "Получено $num_correct правильных ответов на \
$num_ques вопросов."
echo "Ваша оценка в процентах правильных ответов - $percent."
}
#=================
# Main program
initialize                         # Шаг 1 основного замысла
subject=$(choose_subj)             # Шаг 2
[[ $? -eq 0 ]] || exit 2           # Если выбор не верен - выход
```

```
cd $subject || exit 2              # Шаг 3
echo                               # Пропуск строки
scramble                           # Шаг 4
for ques in ${questions[*]}; do    # Шаг 5
    ask $ques
    result=$?
    (( num_ques=num_ques+1 ))
    if [[ $result == 1 ]]; then
        (( num_correct += 1 ))
    fi
    echo                           # Пропуск строки между вопросами
    sleep ${QUIZDELAY:=1}
done
summarize                          # Шаг 6
exit 0
```

Заключение

Оболочка является языком программирования. Программы, написанные на этом языке, называются сценариями оболочки или просто сценариями. Сценарии оболочки предоставляют управляющие структуры для принятия решений и организации циклов, имеющиеся в языках высокого уровня, и вместе с тем предоставляют простой доступ к системным утилитам и пользовательским программам. Для упрощения сложных задач и разбиения их на модули сценарии оболочки могут использовать функции.

Управляющие структуры К управляющим структурам, выбирающим варианты выполнения команд, относятся **if...then**, **if...then...else** и **if...then...elif**. Управляющая структура **case** предоставляет возможность множественного разветвления хода выполнения программы и может быть использована при необходимости выразить альтернативы с использованием простого синтаксиса соответствия шаблону.

К циклическим управляющим структурам относятся **for...in**, **for**, **until** и **while**. Эти структуры осуществляют повторное выполнение одной или нескольких задач.

Инструкции **break** и **continue** изменяют управление ходом программы внутри циклов: инструкция **break** передает управление за пределы цикла, а инструкция **continue** передает управление сразу же на вершину цикла.

Структура **Here document** позволяет вводить в команду сценария оболочки данные, поступающие непосредственно из самого сценария.

Описатели файлов Оболочка Bourne Again Shell предоставляет возможность работы с описателями файлов. В сочетании со встроенными командами **read** и **echo**, описателя файлов позволяют получить такую же степень управления вводом и выводом, которая предоставляется программам, написанным на языках низкого уровня.

Переменные Встроенная команда **typeset** присваивает переменным оболочки **bash** такие атрибуты, как `readonly` (только для чтения). Оболочка Bourne Again Shell предоставляет операторы для обнаружения переменных, соответствующих шаблонам, предоставляет для переменных значения по умолчанию и вычисляет длину переменных. Эта оболочка также поддерживает переменные в виде массивов и ло-

кальные переменные для функций и предоставляет встроенные арифметические операции над целыми числами с использованием встроенной команды let и синтаксиса выражений, похожего на тот, что используется в языке программирования C.

Встроенные команды

В перечень встроенных команд оболочки Bourne Again Shell входят такие команды, как type, read, exec, trap, kill и getopts. Встроенная команда type выводит информацию о команде, включая ее местонахождение, а встроенная команда read позволяет сценарию принимать данные, веденные пользователем.

Встроенная команда exec выполняет команду без создания нового процесса. Новая команда замещает текущий процесс, принимая среду окружения и PID-номер этого процесса. Эта встроенная команда выполняет пользовательские программы и другие команды Linux в том случае, когда нет необходимости возвращать управление вызывавшему их процессу.

Встроенная команда trap перехватывает сигнал, отправленный операционной системой Linux тому процессу, который запустил сценарий, и позволяет указать действия, которые должны быть выполнены после получения одного или нескольких сигналов. Эту встроенную команду можно использовать для того, чтобы заставить сценарий проигнорировать посланный ему сигнал, когда пользователь нажимает клавишу прерывания.

Встроенная команда kill завершает выполнение запущенной программы. А встроенная команда getopts производит разбор аргументов командной строки, упрощая написание программ, которые придерживаются стандартных соглашений, принятых в Linux в отношении аргументов и ключей командной строки.

Утилиты в сценариях

Кроме использования управляющих структур, встроенных команд и функций, сценарии оболочки обычно вызывают утилиты Linux. Например, утилита find часто используется в сценариях, которые осуществляют поиск файлов в иерархии системы, и способна выполнять широкий диапазон задач, от самых простых до самых сложных.

Выражения

Существует два основных типа выражений: арифметические и логические. Арифметические выражения позволяют осуществлять арифметические действия над константами и переменными, возвращая числовой результат. Логические (булевы) выражения сравнивают выражения или строки или проверяют условия для возвращения истинного или ложного результата — *true* или *false*. Как и во всех остальных решениях, возвращаемых в сценариях оболочки Linux, статус *true* представлен нулевым значением, а статус *false* — любым ненулевым значением.

Хороший стиль программирования

В качественно написанных сценариях оболочки соблюдаются определенные стандарты программирования, к которым относятся указание в первой строке сценария той оболочки, которая должна его выполнить, проверка количества и типов аргументов, с которыми вызван сценарий, вывод стандартного сообщения о порядке использования при сообщении об ошибках, допущенных в командной строке, и перенаправление всех информационных сообщений на стандартную ошибку.

Упражнения

1. Перепишите сценарий journal из упражнения 5 главы 8, добавив команды для проверки наличия у пользователя права записи в файл journal-file, который находится в его личном каталоге. Сценарий должен выполнять соответствующие действия, если файл journal-file существует и у пользователя нет права записи в этот файл. Проверьте работоспособность измененного сценария.
2. В сценарии out (см. раздел «Управляющая структура if...then...else» на с. 421) имеются две ссылки на специальный параметр "$@". Объясните, что изменится, если вместо этого параметра будет использован параметр "$*".
3. Напишите фильтр, в который вводится список файлов и из которого выводится простое имя (см. подраздел «Имя программы» в разделе «Управляющая структура case» на с. 444 каждого файла, присутствующего в этом списке.
4. Напишите функцию, которая принимает в качестве аргумента имя файла и добавляет к этому файлу право пользователя на его выполнение.
   1) Где может пригодиться такая функция?
   2) Внесите в сценарий изменения, чтобы он принимал в качестве аргументов одно или несколько имен файлов и добавлял к каждому файлу, имя которого указано в качестве аргумента, право пользователя на его выполнение.
   3) Что можно сделать, чтобы эта функция была доступна после каждого входа в систему?
   4) Предположим, что в дополнение к доступности функции при последующих сеансах работы нужно сделать эту функцию доступной в вашей текущей оболочке. Как это сделать?
5. Когда может появиться необходимость или желание написать сценарий вместо функции оболочки? Придумайте как можно больше причин для этого.
6. Напишите сценарий оболочки, который выводит имена всех файлов каталогов в рабочем каталоге, игнорируя все остальные типы файлов.
7. Напишите сценарий, который выводит время каждые 15 секунд. Прочитайте man-страницу, посвященную команде date, и выведите время, используя описатель поля %r. Очищайте окно (используя команду clear) перед каждым выводом времени.
8. Введите следующий сценарий, названный savefiles, и присвойте себе право выполнения файла:

```
$ cat savefiles
#! /bin/bash
echo "Сохранение файлов в текущем каталоге в файле savethem."
exec > savethem
for i in *
        do
        echo "================================================"
        echo "Файл: $i"
        echo "================================================"
        cat "$i"
        done
```

   1) Какое сообщение об ошибке будет получено при выполнении этого сценария? Перепишите сценарий, чтобы не появлялась эта ошибка, сохранив направление вывода в savethem.
   2) С какой проблемой можно столкнуться, если дважды запустить этот сценарий в одном и том же каталоге? Рассмотрите возможность решения данной проблемы.

9. Прочитайте man- или info-страницу, посвященную bash, проведите ряд экспериментов и ответьте на следующие вопросы:
   1) Как экспортируется функция?
   2) Чем занимается встроенная команда hash?
   3) Что произойдет, если отсутствует право на выполнение файла, чье имя указано команде exec в качестве аргумента?
10. Используя утилиту find, выполните следующие задачи:
   1) Выведите список всех файлов рабочего каталога и всех подкаталогов, подвергшихся изменениям за последний день.
   2) Выведите список всех файлов, в отношении которых имеется право на чтение, чей размер больше одного мегабайта.
   3) Удалите все файлы по имени core из структуры каталогов, берущей начало в вашем личном каталоге.
   4) Выведите список номеров i-узлов всех файлов в рабочем каталоге, чьи имена заканчиваются на .c.
   5) Выведите список всех файлов, принадлежащих файловой системе корневого каталога, которые подвергались изменениям за последние 30 суток.
11. Напишите небольшой сценарий, сообщающий об идентичности прав доступа к двум файлам, чьи имена заданы этому сценарию в качестве аргументов. Если права доступа к этим двум файлам идентичны, выведите общее поле прав доступа. В противном случае выведите имя каждого файла, за которым следует поле прав доступа к нему. (*Подсказка.* Попробуйте воспользоваться утилитой cut.)
12. Напишите сценарий, получающий в качестве аргумента имя каталога и проводящий поиск файлов нулевой длины в иерархии файлов, берущей начало в этом каталоге. Запишите имена всех файлов нулевой длины на стандартный вывод. При отсутствии ключей в командной строке заставьте сценарий удалить файл после вывода его имени, если после запроса подтверждения пользователь дал положительный ответ. Ключ -f (force — принудительно) в командной строке должен служить признаком того, что сценарий должен отобразить имя файла, но не должен запрашивать подтверждение перед его удалением.

Упражнения повышенной сложности

1. Напишите сценарий, который принимает список записей, разделенных двоеточиями, и построчно выводит записи на стандартный вывод (без двоеточий).
2. Придайте сценарию из упражнения 13 более универсальный характер, позволяющий указывать символ-разделитель записей в качестве аргумента сценария. При отсутствии данного аргумента в качестве символа-разделителя по умолчанию должно рассматриваться двоеточие.
3. Напишите функцию по имени funload, которая в качестве своего единственного аргумента принимает имя файла, содержащего другие функции. Цель функции funload состоит в том, чтобы сделать все функции в указанном файле доступными текущей оболочке; то есть функция funload должна загружать функции из указанного файла. Чтобы определить местонахождение файла, функция funload должна вести поиск в каталогах, чьи имена заданы в переменной среды окружения FUNPATH и отделены друг от друга двоеточиями. Предположим, что формат у переменной FUNPATH такой же, как и у переменной PATH, и поиск в FUNPATH похож на тот поиск, который оболочка ведет в переменной PATH.

16. Перепишите сценарий bundle (см. раздел «Структура Here Document» на с. 447), чтобы он принимал в качестве аргументов необязательный список имен файлов. Если в командной строке задано одно или несколько имен файлов, то воссозданы должны быть только эти файлы, в противном случае должны быть воссозданы все файлы архива оболочки. Представьте, к примеру, что в архив по имени srcshell собраны все файлы с расширениями имен .c и нужно воссоздать только файлы test1.c и test2.c. Извлечение из архива только этих двух файлов должно произойти с помощью следующей команды:

```
$ bash srcshell test1.c test2.c
```

17. Какой тип ссылок не обнаруживается сценарием lnks (см. раздел «Управляющая структура if...then...elif» на с. 423)? Почему?

18. В принципе, в любом случае можно обойтись и без рекурсии, заменив ее конструкцией последовательного перебора, например while или until. Перепишите функцию makepath (см. раздел «Рекурсивный сценарий оболочки» на с. 489), превратив ее в нерекурсивную функцию. Какой версии вы отдадите предпочтение? Почему?

19. Списки часто хранятся в переменных среды окружения путем помещения двоеточия (:) между всеми элементами списка. (Примером может послужить значение переменной PATH.) Добавить элемент к такому списку можно путем присоединения нового элемента к началу списка:

```
PATH=/opt/bin:$PATH
```

    Если добавляемый элемент уже присутствует в списке, то теперь там будет две копии этого элемента. Напишите функцию оболочки по имени addenv, принимающую два аргумента: первый — имя переменной оболочки, и второй — строку, которая ставится впереди списка, являющегося значением переменной оболочки только в том случае, если эта строка еще не является элементом списка. Например, вызов

```
addenv PATH /opt/bin
```

    должен привести к добавлению каталога /opt/bin к переменной PATH только в том случае, если такого каталога в ней нет. Обеспечьте работу своего решения даже в том случае, если переменная оболочки изначально была пустой. Также обеспечьте тщательную проверку элементов списка. Если в переменной PATH есть каталог /usr/opt/bin, но нет каталога /opt/bin, то только что приведенный пример должен добавить каталог /opt/bin к переменной PATH. (*Подсказка.* Возможно, с этим упражнением будет легче справиться, если сначала написать функцию locate_field, сообщающую о том, является ли строка элементом значения переменной.)

20. Напишите функцию, которая принимает в качестве аргумента имя каталога и записывает в стандартный вывод максимальное значение длины всех файлов, имеющихся в этом каталоге. Если аргумент функции не является именем каталога, сообщение об ошибке должно быть записано в стандартный вывод, а выход из функции должен быть осуществлен с ненулевым статусом.

21. Измените функцию, написанную для упражнения 20, чтобы осуществлялся спуск по всем подкаталогам указанного каталога в рекурсивном режиме, и поиск максимальной длины файла с любым именем осуществлялся в этой иерархии.

22. Напишите функцию, которая выводит список количества обычных файлов, каталогов, специальных блочных файлов, специальных символьных файлов, файлов FIFO и символьных ссылок, имеющихся в рабочем каталоге. Сделайте это двумя разными способами:

    1) Используйте для определения типа файла первую букву вывода команды `ls -l`.

    2) Используйте для определения типа файла условные тесты с синтаксисом *[[выражение]]*.

23. Измените программу quiz (см. раздел «Сценарий оболочки quiz» на с. 492), чтобы выборы ответов на вопросы были выстроены в случайном порядке.

Глава 11. Язык сценариев Perl

Ларри Уолл (Larry Wall) создал Perl (Practical Extraction and Report Language — практический язык извлечений и отчетов) для работы с текстом. В языке Perl используются синтаксис и понятия, позаимствованные из awk, sed, C, оболочки Bourne Shell, языков Smalltalk, Lisp и из обычного английского языка. Этот язык был разработан для извлечения информации из текстовых файлов и создания отчетов на основе этой информации. Со времени своей официальной презентации в 1987 году, Perl был существенно расширен, вместе с ним пополнялась и документация по этому языку. В наши дни, кроме обработки текста, Perl используется для системного администрирования, разработки программного обеспечения и программирования общего назначения.

Код Perl обладает хорошей переносимостью, поскольку язык Perl был реализован на многих операционных системах (см. www.cpan.org/ports). Perl является вполне естественным, удобным, понятным, простым в использовании, эффективным и самодостаточным языком. Он считается простым и понятным языком, поддерживающим процедурное и объектно-ориентированное программирование, но при этом лишенным особой элегантности.

Одним из важных отличий Perl от многих других языков программирования являются его лингвистические истоки. По-английски можно сказать «I will buy a car if I win the lottery» («Я куплю машину, если выиграю в лотерею»). Perl дает пользователю возможность подражать такому синтаксису. Другим отличием является то, что в Perl имеются сингулярные и множественные переменные, первые из них содержат одиночные значения, а вторые — списки значений.

Введение в Perl

Свет на философию Perl проливают две цитаты из руководства пользователя:

> *Многие элементы синтаксиса Perl являются необязательными. Вместо того чтобы выполнить требования по использованию круглых скобок вокруг каждого вызова функции и по объявлению каждой переменной, во многих случаях такие явно обозначенные элементы можно опустить, и Perl, чаще всего, сможет понять, что именно под этим всем подразумевалось. Подобное поведение известно как Do What I Mean, или сокращенно — DWIM, что означает «Делай то, что я имею в виду». Данное свойство позволяет программистам использовать только самое необходимое и программировать в удобном для них стиле.*
>
> *Девиз Perl — «There's more than one way to do it» («Всегда есть несколько способов решения любой задачи»). Можно только догадываться, сколько новых открытий в использовании этого языка может сделать для себя читатель.*

Одно из самых ценных качеств языка Perl заключается в том, что он поддерживается тысячами модулей, созданных независимыми разработчиками. Всеобъемлющая сеть архивов Perl — The Comprehensive Perl Archive Network (CPAN; www.cpan.org), является хранилищем множества модулей и дополнительной информации, имеющей

отношение к Perl. Сведения о загрузке, установке и использовании этих модулей в программах на языке Perl даны в разделе «Модули CPAN» на с. 544.

УСТАНОВКА ПАКЕТА PERL-DOC

Пакет perl-doc содержит большой объем информации. Установите этот пакет перед началом использования Perl, далее о нем будет дана дополнительная информация.

Лучший способ изучения Perl заключается в работе на этом языке. Копируйте программы из этой главы и вносите в них изменения, пока они не обретут для вас вполне определенный смысл. На Perl написаны многие системные инструментальные средства. Первые строки большинства этих средств начинаются с оповещения #!/usr/bin/perl, заставляющего оболочку передавать программу на выполнение интерпретатору языка Perl. Многие файлы, содержащие строку /usr/bin/perl, являются программами на языке Perl. В следующей команде для рекурсивного (-r) поиска файлов, содержащих строку /usr/bin/perl в каталогах /usr/bin и /usr/sbin, используется утилита grep, которая выводит список многих локальных системных средств, написанных на Perl:

```
$ grep -r /usr/bin/perl /usr/bin /usr/sbin | head -4
/usr/bin/defoma-user:#! /usr/bin/perl -w
/usr/bin/pod2latex:#!/usr/bin/perl
/usr/bin/pod2latex:    eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
/usr/bin/splain:#!/usr/bin/perl
```

Просмотрите эти программы, чтобы понять, как используется Perl в реальных программных средствах. Перед внесением изменений в системную программу скопируйте ее в один из своих каталогов. Не запускайте системную программу при работе с root-правами, если не до конца представляете себе, что из этого может получиться.

Дополнительная информация

Локальная — man-страницы: просмотрите man-страницы perl и perltoc, в которых приводятся списки man-страниц, посвященных Perl

Из Интернета — Главная страница Perl: www.perl.com

CPAN: www.cpan.org

блог: perlbuzz.com

Книги — «Programming Perl», третье издание, Уолл (Wall), Кристиансен (Christiansen) и Орвант (Orwant), O'Reilly & Associates, июль 2000. Русский перевод вышел в 2006 году.

Помощь

Perl — язык, прощающий многие ошибки. Поэтому нетрудно написать программу на Perl, которая будет работать, но при этом делать совсем не то, для чего она предназначалась. В Perl имеется множество инструментальных средств, которые могут оказать помощь в поиске ошибок программирования. Полезные диагностические сообщения можно получить с помощью ключа -w и использования предупреждающих инструкций. Использование инструкции strict (см. man-страницу perldebtut) поможет навести порядок в программе за счет выставления, кроме всего прочего, требования по

объявлению переменных перед их использованием. Если причина неудачной работы заключается в чем-нибудь другом, можно воспользоваться встроенным отладчиком Perl и запустить пошаговое выполнения программы. Дополнительные сведения можно найти на man-страницах perldebtut и perldebug.

Утилита perldoc

Перед тем как использовать perldoc, нужно установить пакет perl-doc.

Утилита perldoc находит и отображает локальную документацию по языку Perl. Она похожа на утилиту man (см. раздел «man: Отображение системного руководства» на с. 53), но относится к языку Perl. Она работает с файлами, которые включают строки на простом и понятном языке создания документации — pod (plain old documentation — простая документация). При встраивании в программу Perl pod позволяет включать документацию для всей программы, а не только комментарии на уровне кода.

Следующий код представляет собой простую программу на Perl, включающую pod. Две строки, следующие за строкой =cut, являются программой, а все остальное является документацией в формате pod.

```
$ cat pod.ex1.pl
#!/usr/bin/perl

=head1 Perl-программа, выводящее слово I<Привет.>

Эта простая Perl-программа включает документацию в формате B<pod>.
Следующая команда B<=cut> сообщает B<perldoc>, что все остальное
не является документацией.

=cut
# Программа на Perl
print "Привет.\n";

=head1 pod Краткая документация по любой команде pod

Дополнительные сведения о B<pod> - на странице B<perldoc.perl.org/perlpod.html>,
а полная документаци по Perl - на странице B<perldoc.perl.org>.
```

Для запуска программы можно использовать Perl:

```
$ perl pod.ex1.pl
Hi there.
```

А для вывода документации можно воспользоваться утилитой perldoc:

```
$ perldoc pod.ex1.pl
POD.EX1(1)            User Contributed Perl Documentation           POD.EX1(1)
Perl-программа, выводящее слово Привет.
    Эта простая Perl-программа включает документацию в формате pod.
    Следующая команда =cut сообщает perldoc, что все остальное
    не является документацией.

pod Краткая документация по любой команде pod
    Дополнительные сведения о pod - на странице perldoc.perl.org/perlpod.html,
    а полная документаци по Perl - на странице perldoc.perl.org.
perl v5.10.0                  2008-10-14                        POD.EX1(1)
```

Наиболее распространенные модули и сценарии, как, собственно, и сам Perl, включают встроенную документацию в pod-формате. Например, следующая команда выводит информацию об имеющейся в Perl функции print:

```
$ perldoc -f print
    print FILEHANDLE LIST
    print LIST
    print Prints a string or a list of strings. Returns true if success-
            ful.  FILEHANDLE may be a scalar variable name, in which case
          the variable contains the name of or a reference to the file-
          handle, thus introducing one level of indirection.  (NOTE: If
          FILEHANDLE is a variable and the next token is a term, it may
          ...
```

Как только будет установлен какой-нибудь модуль (см. раздел «Модули CPAN» на с. 544), для вывода документации по нему можно будет воспользоваться утилитой perldoc. В следующем примере показана работа perldoc по выводу информации об установленном в локальной системе модуле Timestamp::Simple:

```
$ perldoc Timestamp::Simple
Timestamp::Simple(3)  User Contributed Perl Documentation Timestamp::Simple(3)

NAME
      Timestamp::Simple - Simple methods for timestamping
SYNOPSIS
          use Timestamp::Simple qw(stamp);
          print stamp, "\n";
...
```

Чтобы вывести man-страницу, посвященную perldoc, и прочитать дополнительную информацию об этой утилите, нужно ввести команду `man perldoc` или `perldoc perldoc`.

ПОДУМАЙТЕ ОБ УДОБСТВЕ ЧТЕНИЯ ПРОГРАММ НА PERL

Несмотря на наличие в Perl множества сокращений, как нельзя лучше подходящих для программирования на скорую руку, в этой главе представлен код, в котором нетрудно разобраться и который несложно поддерживать.

Терминология

В этом разделе определены некоторые понятия, используемые в данной главе.

Модуль

Модуль в Perl является самодостаточным фрагментом кода, который чаще всего состоит из нескольких взаимодействующих функций. Модуль может быть вызван из другого модуля или из программы на языке Perl. У модуля должно быть уникальное имя. Для обеспечения уникальности имен Perl предоставляет для модулей иерархическое пространство имен, где компоненты имени отделяются друг от друга двойными двоеточиями (::). Примерами имен модулей могут послужить Time-stamp::Simple и WWW::Mechanize.

Дистрибутив

Дистрибутив Perl — это набор из одного или нескольких модулей, выполняющих определенную задачу. Дистрибутивы и модули можно найти на сайте search.cpan.org. В качестве примеров дистрибутивов можно привести Timestamp-Simple (в архивном файле Timestamp-Simple-1.01.tar.gz содержится лишь

один модуль Timestamp::Simple) и WWW-Mechanize (в файле WWW-Mechanize-1.34.tar.gz содержится модуль WWW::Mechanize, плюс поддерживающие модули, в числе которых WWW::Mechanize::Link и WWW::Mechanize::Image).

Пакет

Пакет определяет пространство имен Perl. Например, в переменной по имени `$WWW::Mechanize::ex` `$ex` — скалярная переменная из пакета WWW::Mechanize, где слово «пакет» используется в смысле пространства имен. Использование одного и того же имени, к примеру, WWW::Mechanize, для дистрибутива, пакета и модуля может внести ненужную путаницу.

Блок

Блок состоит из нуля и более инструкций, заключенных в фигурные скобки ({}), которые определяют область видимости. При объяснении управляющих структур оболочки эти элементы назывались командами. В качестве примера можно взять управляющую структуру if...then из раздела «Управляющие структуры» на с. 417.

Пакетная переменная

Пакетная переменная определяется внутри пакета, в котором она появляется. Другие пакеты могут ссылаться на пакетные переменные путем использования полного имени переменной (например, `$Text::Wrap::columns`). По умолчанию переменные являются пакетными, пока они не будут определены как лексические.

Лексическая переменная

Лексическая переменная, определяемая путем установки перед именем переменной ключевого слова my (см. совет в разделе «Переменные» на с. 512). Эта переменная определяется только внутри блока или файла, в котором она появляется. В других языках программирования лексические переменные называются локальными. Поскольку в Perl 4 ключевое слово local используется в другом смысле, вместо него в Perl 5 используется ключевое слово lexical. При программировании с использованием оболочки bash те переменные, которые не были экспортированы (см. раздел «Локальность переменных» на с. 454), являются локальными для той программы, в которых они используются.

Список

Список — это комплект, состоящий из нуля и более скаляров. Следующий список содержит три элемента — два числа и строку:

```
(2, 4, 'Zach')
```

Массив

Массив — это переменная, в которой содержится список элементов, имеющий определенный порядок. В следующей строке кода переменная `@a` является массивом. Дополнительные сведения о переменных, являющихся массивами, даны в разделе «Переменные, представляющие собой массивы» на с. 453.

```
@a = (2, 4, 'Zach')
```

Составная инструкция

Составная инструкция — это инструкция, составленная из других инструкций. Например, составная инструкция if (см. раздел «Управляющие структуры» на с. 520) включает саму инструкцию if, в которую обычно включаются другие инструкции, находящиеся внутри управляемого ею блока.

Запуск программы на Perl

Для запуска программы, написанной на языке Perl, существует несколько способов. Ключ -e позволяет вводить программу в командной строке:

```
$ perl -e 'print "Привет.\n"'
Привет.
```

Использование ключа -e дает хорошую возможность для тестирования синтаксиса Perl и запуска небольших, написанных на скорую руку программ. Этот ключ требует, чтобы программа на Perl фигурировала в качестве единого аргумента командной строки. Программа должна следовать сразу же за этим ключом, поскольку она является его аргументом. Простой способ написания подобных программ предусматривает заключение всей программы в одинарные кавычки.

Поскольку Perl относится к классу утилит, которые получают информацию из файла или из стандартного ввода (см. раздел «Утилиты, воспринимающие ввод из файла или со стандартного устройства ввода» на с. 148), можно дать команду perl и ввести программу, завершив ее нажатием комбинации CONTROL+D (конец файла). Perl прочитает программу со стандартного ввода:

```
$ perl
print «Привет.\n";
CONTROL+D
Привет.
```

Предыдущая технология подходит для наскоро набранных одноразовых программ командной строки, но совсем не годится для запуска более сложных программ. Чаще всего, программы на Perl хранятся в текстовых файлах. Хотя это и не требуется, но обычно эти файлы имеют расширение имени .pl. Следующий код представляет ту же простую программу из предыдущего примера, сохраненную в файле:

```
$ cat simple.pl
print "Привет.\n";
```

Эту программу можно запустить, указав ее имя в качестве аргумента для Perl:

```
$ perl simple.pl
Привет.
```

Чаще всего, подобно многим сценариям оболочки, файл, содержащий программу на Perl, является исполняемым. В следующем примере команда chmod (см. раздел «chmod: Превращение файла в исполняемый» на с. 292) делает файл simple2.pl исполняемым. Согласно объяснениям, данным в разделе «#! — способ указания оболочки» на с. 294, символы #! в начале первой строки файла указывают оболочке на необходимость передачи всего остального файла для исполнения программе /usr/bin/perl.

```
$ chmod 755 simple2.pl
$ cat simple2.pl
#!/usr/bin/perl -w
print "Привет.\n";

$ ./simple2.pl
Привет.
```

В данном примере программа simple2.pl выполняется как ./simple2.pl, поскольку рабочий каталог не фигурирует в значении переменной PATH данного пользователя (см. раздел «Переменная PATH: Указание тех мест, где оболочка ищет программы» на с. 312). Ключ -w заставляет Perl выводить предупреждения при обнаружении в коде потенциальных ошибок.

Perl версии 5.10

Все примеры данной главы запускались под управлением Perl 5.10. Чтобы увидеть, какая версия Perl запущена на вашей локальной системе, нужно ввести следующую команду:

```
$ perl -v
This is perl, v5.10.0 built for i486-linux-gnu-thread-multi
...
```

Использование свойства say (use feature 'say')

Функция say является свойством Perl 6, доступным в Perl 5.10. Эта функция работает точно так же, как и функция print, за исключением того, что она добавляет разделитель строк (\n) к концу каждой выводимой строки. Некоторые версии Perl требуют специального оповещения о намерениях использования функции say. В следующем примере с помощью функции use интерпретатору Perl разрешается использование функции say. Попробуйте запустить эту программу без строки с вызовом функции use, чтобы посмотреть, нужна ли она для локальной версии Perl.

```
$ cat 5.10.pl
use feature 'say';
say 'Вывод с помощью say.';
print 'Вывод с помощью print.';
say 'Конец.'
$ perl 5.10.pl
Вывод с помощью say.
Вывод с помощью.Конец.
$
```

Более ранние версии Perl

Если у вас запущена более ранняя версия Perl, нужно будет заменить инструкцию say в примерах данной главы на инструкцию print и завершить эту инструкцию процитированным символом \n:

```
$ cat 5.8.pl
print 'Вывод с помощью print вместо say.', "\n";
print 'Вывод с помощью print.';
print 'Конец.', "\n";

$ perl 5.8.pl
Вывод с помощью print вместо say.
Вывод с помощью print.Конец.
```

Синтаксис

В этом разделе рассматриваются основные компоненты программы на Perl.

Инструкции

Программа на Perl состоит из одной или нескольких *инструкций*, каждая из которых завершается точкой с запятой (;). В этих инструкциях можно использовать любое количество *пробельных символов*, за исключением тех, которые находятся внутри закавыченных строк. В одной строке могут находиться сразу несколько инструкций, каждая из которых должна быть завершена точкой с запятой. Следующие программы функционально похожи друг на друга. Первая из них

занимает две строки, а вторая только одну. Обратите внимание на разницу в присутствии пробелов вокруг знака равенства и знака плюс. Если программа жалуется на недоступность функции say, обратитесь к расположенному выше разделу «Использование свойства say (use feature 'say')» на с. 510.

```
$ cat statement1.pl
$n=4;
say "Ответ ", $n + 2;
$ perl statement1.pl
Ответ 6

$ cat statement2.pl
$n = 4; say "Ответ ", $n+2;
$ perl statement2.pl
Ответ 6
```

Выражения Синтаксис выражений Perl часто перекликается с синтаксисом выражений языка C, но между ними имеется ряд отличий. Выражения Perl рассмотрены в примерах данной главы.

Кавычки Все символьные строки должны быть заключены в одинарные или двойные кавычки. Perl различает эти два типа кавычек примерно так же, как это делают оболочки (см. раздел «Пользовательские переменные» на с. 306): двойные кавычки позволяют Perl осуществлять подстановку значений заключенных в них переменных и интерпретировать специальные символы, например \n (разделитель строк), а одинарные кавычки этого делать не позволяют. Некоторые специальные символы Perl перечислены в табл. 11.1.

В следующем примере показано, как различные типы кавычек (и их отсутствие) влияют на Perl при преобразовании скалярных переменных из чисел в строки и обратно. В первой инструкции print одинарные кавычки препятствуют тому, чтобы Perl осуществлял подстановку значения переменной $string, и тому, чтобы он интерпретировал специальный символ \n. Первый символ \n во второй инструкции print заставляет эту инструкцию осуществлять вывод с новой строки.

```
$ cat string1.pl
$string="5";      # $string объявляется строковой, но это не имеет значения

print '$string+5\n';   # Perl выводит $string+5 в буквальном виде, потому что
                       # выражение взято в одинарные кавычки
print "\n$string+5\n"; # Perl подставляет значение $string как строку,
                       # потому что используются двойные кавычки
print $string+5, "\n"; # Отсутствие кавычек заставляет Perl интерпретировать
                       # $string как числовую переменную и прибавить к ней 5;
                       # Символы \n должны быть в двойных кавычках

$ perl string1.pl
$string+5\n
5+5
10
```

Слэш По умолчанию регулярные выражения ограничиваются знаками слэша (/). В следующем примере проверяется, не содержит ли строка hours

шаблон our; дополнительные сведения о регулярных выражениях Perl даны в разделе «Символ / как ограничитель по умолчанию» на с. 538.

```
$ perl -e 'if ("hours" =~ /our/) {say "yes";}'
```

Для нормальной работы в локальной версии Perl, может понадобиться задействовать свойство say (см. раздел «Использование свойства say (use feature 'say')» на с. 510):

```
$ perl -e 'use feature "say"; if ("hours" =~ /our/) {say "yes";}'
```

Обратный слэш Внутри строки, взятой в двойные кавычки, обратный слэш выключает (цитирует) другой обратный слэш. Поэтому Perl выводит "\\n" как \n. Внутри регулярного выражения Perl не расширяет метасимвол, перед которым стоит обратный слэш (см. пример string1.pl на с. 511).

Комментарии Как и в оболочке, комментарии в Perl начинаются со знака решетки (#) и заканчиваются концом строки (непосредственно перед разделителем строк).

Специальные символы В табл. 11.1 перечислены некоторые символы, которые имеют специальное значение в строках Perl. Когда эти символы встречаются в двойных кавычках, Perl производит из замену. В табл. 11.3 перечислены метасимволы, имеющие специальное значение внутри регулярных выражений.

Таблица 11.1. Некоторые специальные символы Perl

Символы	Чем заменяются внутри двойных кавычек
\0*xx* (нуль)	ASCII-символом, чье восьмеричное значение равно *xx*
\a	Символом сигнала (bell или beep) с ASCII- кодом 7
\e	Символом ESCAPE (ASCII-код 27)
\n	Символом разделителя строк, NEWLINE (ASCII-код 10)
\r	Символом RETURN (ASCII-код 13)
\t	Символом TAB (ASCII-код 9)

Переменные

Как и в обычных языках общения, Perl различает единственное и множественное числа. Строки и числа относятся к единственному числу, а списки строк или чисел — ко множественному. Perl предоставляет три типа переменных: скаляры (единственного числа), массивы (множественного числа) и хеши (именуемые также ассоциативными массивами, также множественного числа). Perl задает каждый тип переменной с помощью специального символа, который ставится перед ее именем. Имя скалярной переменной начинается со знака доллара ($), переменной массива — со знака at (@), а переменной хеша — со знака процента (%). В отличие от способа идентификации переменных оболочками, Perl требует наличия этого спецсимвола при каждой ссылке на переменную, включая случай присваивания ей значения:

```
$ name="Zach" ; echo "$name"                    (bash)
Zach

$ perl -e '$name="Zach" ; print "$name\n";'     (perl)
Zach
```

В имена переменных, небезразличных к регистру используемых букв, могут включаться буквы, цифры и знак подчеркивания (_). Переменная Perl является пакетной переменной (см. ранее данное определение), пока перед ней не будет стоять ключевое слово my, тогда она будет считаться лексической переменной (см. ранее данное определение), то есть переменной, которая определена внутри блока или файла, в котором она появляется. Локальность переменных Perl рассмотрена в разделе «Подпрограммы» на с. 535.

ЛЕКСИЧЕСКИЕ ПЕРЕМЕННЫЕ ЗАКРЫВАЮТ СОБОЙ ПАКЕТЫЕ ПЕРЕМЕННЫЕ

Если лексическая переменная и пакетная переменная имеют одно и то же имя, то внутри блока или файла, в котором определена лексическая переменная, это имя является ссылкой на лексическую, а не на пакетную переменную.

Переменные Perl начинают свое существование с присвоением им значения, их не нужно объявлять или инициализировать, хотя это могло бы сделать программу более понятной. Обычно Perl не жалуется на ссылку на неинициализированную переменную:

```
$ cat variable1.pl
#!/usr/bin/perl
my $name = 'Sam';
print "Привет, $nam, как дела?\n";          # Опечатка, не допечатана 'e' в слове name

$ ./variable1.pl
Привет, , как дела?
```

use strict

Чтобы заставить Perl требовать объявления переменных перед присваиванием им значений, в программу включается инструкция `use strict`. Дополнительная информация об этой инструкции дана на man-странице **perldebtut**. При включении в предыдущую программу строки `use strict` Perl выводит сообщение об ошибке:

```
$ cat variable1b.pl
#!/usr/bin/perl
use strict;
my $name = 'Sam';
print "Привет, $nam, как дела?\n"; # Опечатка, не допечатана 'e' в слове name

$ ./variable1b.pl
Global symbol "$nam" requires explicit package name at ./variable1b.pl line 4.
Execution of ./variable1b.pl aborted due to compilation errors.
Глобальный символ "$nam" требует явного указания имени пакета в ./variable1b.pl
строка 4.
Выполнение ./variable1b.pl прервано из-за ошибок компиляции.
```

ИСПОЛЬЗОВАНИЕ MY: СРАВНЕНИЕ ЛЕКСИЧЕСКИХ И ПАКЕТНЫХ ПЕРЕМЕННЫХ

В программе variable1.pl, переменная $name объявлена лексической путем установки перед ее именем ключевого слова my; ее имя и значение известны только в пределах файла variable1.pl. Объявление переменной лексической ограничивает ее область видимости тем блоком или файлом, в котором она определена. Хотя в данном случае этого и не нужно, объявление переменных лексическими является весьма полезной привычкой. Особенно полезно придерживаться этой привычки при написании больших программ, подпрограмм и пакетов,

где становится все труднее сохранять уникальность имен переменных. Объявление всех переменных лексическими становится обязательным при написании подпрограмм, которые будут использоваться в составе кода, который написан другими программистами. Это позволит всем, кто работает с вашими подпрограммами, использовать имена переменных по своему усмотрению, независимо от того, какие имена использовались в написанном вами коде.

У оболочки и у Perl области видимости переменных определяются по-разному. В оболочке, если переменная не экспортирована, она является локальной по отношению к использующей ее подпрограмме (см. раздел «Локальность переменных» на с. 454). В Perl, если для объявления переменной не используется ключевое слово my, чтобы превратить ее в лексическую, она определяется для того пакета, в котором появляется.

Ключ –w и инструкция use warnings

Ключ -w и инструкция use warnings выполняют одинаковые функции: они заставляют Perl генерировать сообщение при обнаружении синтаксической ошибки. В следующем примере Perl выводит два предупреждения. Первое сообщает о том, что переменная по имени $nam использовалась один раз в строке 3, что, возможно, свидетельствует об ошибке. Это сообщение может оказаться полезным при опечатке, допущенной в имени переменной. В Perl 5.10 второе предупреждение указывает имя неинициализированной переменной. Это предупреждение ссылается на ту же проблему, что и первое. Хотя в этой простой программе совсем нетрудно определить, какая из двух переменных оказалась неинициализированной, в более сложной программе это может занять уйму времени.

```
$ cat variable1a.pl
#!/usr/bin/perl -w
my $name = 'Sam';
print "Привет, $nam, как дела?\n"; # Вывод предупреждения (опечатка и ключ -w)

$ ./variable1a.pl
Name "main::nam" used only once: possible typo at ./variable1a.pl line 3.
Имя "main::nam" используется только один раз: возможно, опечатка в строке 3
в ./variable1a.pl.
Use of uninitialized value $nam in concatenation (.) or string at ./variable1a.pl
line 3.
Использование неинициализированной переменной $nam в конкатенации (.) или в строке
в файле ./variable1a.pl, в строке 3.
Привет, , как дела?
```

Ключ -w можно также использовать в командной строке. Если вместе с ним используется ключ -e, то сразу за ним в качестве аргумента должна следовать программа, которую необходимо выполнить (то есть сочетание -e -w работать не будет). См. совет «Использование других ключей вместе с ключом –e» на с. 539.

```
$ perl -w -e 'my $name = "Sam"; print "Hello, $nam, how are you?\n"'
Name "main::nam" used only once: possible typo at -e line 1.
Имя "main::nam" используется только один раз: возможно, опечатка в -e, в строке 1
Use of uninitialized value $nam in concatenation (.) or string at -e line 1.
Использование неинициализированной переменной $nam в конкатенации (.) или в строке
в -e, в строке 1
Привет, , как дела?
```

undef и defined

Неопределенная переменная имеет специальное значение undef, которое в числовых выражениях вычисляется в нуль (0), а при выводе расширяется в пустую строку (""). Чтобы определить, была ли переменная определена,

используется функция defined. В следующем примере, в котором используется конструкция, рассматриваемая далее в данной главе, функция defined вызывается с аргументом $name, а результат инвертируется с помощью восклицательного знака (!). В результате получается, что инструкция print выполняется в том случае, если переменная $name не определена.

```
$ cat variable2.pl
#!/usr/bin/perl
if (!defined($name)) {
print "Переменная '\$name' не определена.\n"
};

$ ./variable2.pl
Переменная '$name' не определена.
```

Поскольку ключ -w заставляет Perl выдавать предупреждения при ссылках на неопределенную переменную, использование этого ключа вызывает генерацию предупреждения.

Скалярные переменные

Имя скалярной переменной начинается со знака доллара ($), а в самой переменной содержится одна строка или число, поскольку она относится к сингулярным переменным. Так как при необходимости Perl проводит преобразование между двумя типами данных, строки и числа можно использовать на взаимозаменяемой основе. Perl интерпретирует скалярные переменные как строки, если для этого имеется достаточно оснований, и как числа, если есть смысл интерпретировать их в этом качестве. Обычно Perl разбирается в таких вещах довольно неплохо.

В следующем примере показаны варианты использования скалярных переменных. Первые две строки кода (3 и 4) присваивают строку Sam скалярной переменной $name, и числа 5 и 2 скалярным переменным $n1 и $n2 соответственно. В этом примере в одной строке используется несколько инструкций, каждая из которых заканчивается точкой с запятой (;). Если эта программа сообщает о недоступности функции, обратитесь к разделу «Использование свойства say (use feature 'say')» на с. 510.

```
$ cat scalars1.pl
#!/usr/bin/perl -w

$name = "Sam";
$n1 = 5; $n2 = 2;

say "$name $n1 $n2";
say "$n1 + $n2";
say '$name $n1 $n2';
say $n1 + $n2, " ", $n1 * $n2;
say $name + $n1;

$ ./scalars1.pl
Sam 5 2
5 + 2
$name $n1 $n2
```

```
7 10
Argument "Sam" isn't numeric in addition (+) at ./scalers1.pl line 11.
Аргумент "Sam" не является числовым в операции сложения (+) в ./scalers1.pl
в строке 11
5
```

Двойные кавычки

Первая инструкция say отправляет строку, взятую в двойные кавычки, на стандартный вывод (то есть на экран, если не было перенаправления). Внутри двойных кавычек Perl расширяет имена переменных в значения указанных переменных. Поэтому первая инструкция say выводит значения трех переменных, отделенные друг от друга пробелами. Вторая инструкция say включает знак плюс (+). Perl не распознает такие операторы, как + внутри любого типа кавычек. Поэтому Perl выводит знак плюс между значениями двух переменных.

Одинарные кавычки

Третья инструкция say отправляет строку, взятую в одинарные кавычки, на стандартный вывод. Внутри одинарных кавычек Perl интерпретирует все символы буквально, поэтому он выводит строку в том же самом виде, в котором она находится между одинарными кавычками.

В четвертой инструкции say операторы не заключены в кавычки, и Perl выполняет сложение и умножение в соответствии с указаниями. Без пробела, заключенного в кавычки, Perl объединил бы два числа (710). Последняя инструкция say пытается сложить строку и число; ключ `-w` заставляет Perl вывести сообщение об ошибке перед тем, как вывести число 5. Это число является результатом сложения значения `Sam`, которое Perl в числовой среде вычисляет как 0, с числом 5 (0 + 5 = 5).

Переменные, представляющие собой массивы

Переменные, представляющие собой массивы, являются упорядоченным контейнером скаляров, чье имя начинается со знака at (@) и чей первый элемент пронумерован как нулевой (индекс массивов начинается с нуля). Поскольку в любом массиве может содержаться от нуля до нескольких скаляров, он относится к множественной переменной. Массивы, в отличие от хешей (см. раздел «Переменные, представляющие собой хеши» на с. 519), имеют упорядоченную структуру. В Perl размер массива растет по мере необходимости. Если будет ссылка на неинициализированный элемент массива, например на элемент, выходящий за его пределы, Perl возвращает значение undef (не определен).

Первая инструкция в следующей программе присваивает переменной массива `@arrayvar` два числовых и одно строчное значение. Поскольку в Perl индексация начинается с нуля, первая инструкция say выводит значение второго элемента массива (элемента с индексом 1). Эта инструкция определяет переменную `$arrayvar[1]` как скалярную (сингулярную), потому что она ссылается только на одно значение. Вторая инструкция say определяет переменную `@arrayvar[1,2]` как список (множественную переменную), потому что она ссылается на несколько значений (на элементы с индексами 1 и 2).

```
$ cat arrayvar1.pl
#!/usr/bin/perl -w
@arrayvar = (8, 18, "Sam");
say $arrayvar[1];
say "@arrayvar[1,2]";
```

```
$ perl arrayvar1.pl
18
18 Sam
```

В следующем примере демонстрируются два способа определения длины массива и предоставляется дополнительная информация по использованию кавычек в инструкциях print. Первая инструкция присваивания в arrayvar2.pl присваивает значения первым шести элементам массива @arrayvar2. Когда имя массива используется в скалярном контексте, Perl вычисляет его в длину этого массива. Вторая инструкция присваивания дает скалярной переменной $num значение, равное количеству элементов в массиве @arrayvar2.

```
$ cat arrayvar2.pl
#!/usr/bin/perl -w
@arrayvar2 = ("apple", "bird", 44, "Tike", "metal", "pike");

$num = @arrayvar2;                          # количество элементов массива
print "Элементов: ", $num, "\n";            # две равноценные инструкции print
print "Элементов: $num\n";

print "Индекс последнего: $#arrayvar2\n";    # индекс последнего элемента массива

$ ./arrayvar2.pl
Элементов: 6
Элементов: 6
Индекс последнего: 5
```

Первые две инструкции print в arrayvar2.pl выводят строку Элементов:, пробел, значение $num и разделитель строк, и в каждой инструкции используется разный синтаксис. Первая из этих инструкций выводит три значения, используя запятые для их разделения в инструкции print. Вторая инструкция print имеет один аргумент и показывает, как Perl расширяет переменную (заменяет переменную ее значением), когда эта переменная заключена в двойные кавычки.

$#массив

Последняя инструкция print в arrayvar2.pl показывает, что Perl вычисляет переменную $#*массив* как индекс последнего элемента в массиве по имени *массив*. Поскольку Perl по умолчанию использует индексацию, начинающуюся с нуля, эта переменная вычисляется на единицу меньше, чем количество элементов массива.

В следующем примере проводится работа с элементами массива и используется точка (.; оператор объединения, или конкатенации, строк). В первых двух строках присваиваются значения четырем скалярным переменным. В третьей строке показывается, что элементам массива можно присваивать значения, используя скалярные переменные, результаты арифметических вычислений и объединения строк. Оператор точка объединяет строки, поэтому Perl вычисляет выражение $va . $vb как строку Sam, объединенную со строкой uel, то есть как Samuel (обратите внимание на вывод последней инструкции print).

```
$ cat arrayvar3.pl
#!/usr/bin/perl -w
$v1 = 5; $v2 = 8;
```

```
$va = "Sam"; $vb = "uel";
@arrayvar3 = ($v1, $v1 * 2, $v1 * $v2, "Max", "Zach", $va . $vb);

print $arrayvar3[2], "\n";         # один элемент массива – скаляр
print @arrayvar3[2,4], "\n";       # два элемента массива  – список
print @arrayvar3[2..4], "\n\n";    # вырезка

print "@arrayvar3[2,4]", "\n";     # список, элементы, разделенные пробелами
print "@arrayvar3[2..4]", "\n\n"; # вырезка, элементы, разделенные пробелами

print "@arrayvar3\n";              # массив, элементы, разделенные пробелами
```

```
$ ./arrayvar3.pl
40
40Zach
40MaxZach

40 Zach
40 Max Zach

5 10 40 Max Zach Samuel
```

Первая инструкция print в arrayvar3.pl выводит третий элемент (элемент с индексом 2) массива @arrayvar3. В этой инструкции вместо символа @ используется символ $, потому что она ссылается на отдельный элемент массива. В следующих инструкциях print используется символ @, поскольку они ссылаются на более чем один элемент. Внутри квадратных скобок, являющихся признаком индекса массива, два разделенных запятой индекса указывают на два элемента массива. К примеру, вторая инструкция print выводит третий и четвертый элементы массива.

Вырезка из массива

Когда индексы двух элементов массива разделены двумя точками (. . ; оператор диапазона), Perl осуществляет подстановку всех элементов диапазона, включая два указанных элемента. Часть массива, включающая элементы, называется *вырезкой*. Третья инструкция print в предыдущем примере выводит элементы с индексами 2, 3 и 4 (третий, четвертый и пятый элементы), как указано с помощью диапазона 2..4. Perl не ставит пробелы между выводимыми элементами.

В инструкции print, когда переменная массива, включая ее индексы, помещается в двойные скобки, Perl ставит пробелы между всеми элементами. Этот синтаксис показан в пятой и шестой инструкциях print в предыдущем примере. Последняя инструкция print выводит содержимое всего массива с элементами, разделенными пробелами.

shift, push, pop и splice

В следующем примере показаны несколько функций, использующихся для работы с массивами. В нем используется массив @colors, инициализированный названиями семи цветов. Функция shift возвращает и удаляет первый элемент массива, функция push добавляет элемент к концу массива, а функция pop возвращает и удаляет последний элемент из массива. Функция splice заменяет элементы массива другим массивом; в данном примере функция splice вставляет массив @ins, начиная с элемента с индексом 1 (со второго элемента), заменяя два элемента массива. Если эта программа сообщает о недоступности функции say, обратитесь к разделу «Использование свойства say (use feature 'say')» на с. 510. Дополнительные сведения о функциях, рассматриваемых в этом абзаце, даны в man-странице perlfunc.

```
$ cat ./shift1.pl
#!/usr/bin/perl -w
@colors = ("red", "orange", "yellow", "green", "blue", "indigo", "violet");
say "                          Вывод массива: @colors";
say "    Вывод и удаление первого элемента: ", shift (@colors);
say "   Вывод оставшихся элементов массива: @colors";
push (@colors, "WHITE");
say "Добавл. эл-та к концу массива и вывод: @colors";
say " Вывод и удаление посл. эл-та массива: ", pop (@colors);
say "   Вывод оставшихся элементов массива: @colors";
@ins = ("GRAY", "FERN");
splice (@colors, 1, 2, @ins);
say "     Удаление 2 и 3 элементов массива: @colors";

$ ./shift1.pl
                       Вывод массива: red orange yellow green blue indigo violet
    Вывод и удаление первого элемента: red
   Вывод оставшихся элементов массива: orange yellow green blue indigo violet
Добавл. эл-та к концу массива и вывод: orange yellow green blue indigo violet WHITE
 Вывод и удаление посл. эл-та массива: WHITE
   Вывод оставшихся элементов массива: orange yellow green blue indigo violet
     Удаление 2 и 3 элементов массива: orange GRAY FERN blue indigo violet
```

Переменные, представляющие собой хеши

Переменная хеша, иногда называемая переменной ассоциативного массива, — это множественная структура данных, которая содержит массив пар ключ–значение. В хеше в качестве ключей (индексов) используются строки, и он оптимизирован с расчетом на быстрое возвращение значений, соответствующих заданным ключам. Каждый ключ должен представлять собой уникальное скалярное значение. В отличие от массивов (см. раздел «Переменные, представляющие собой массивы» на с. 516) хеши имеют неупорядоченную структуру. Если присвоить хеш списку, пары ключ–значение сохранятся, но их порядок будет отличаться как от алфавитного, так и от того, в котором они вставлялись в хеш, он будет абсолютно произвольным.

Для присваивания значений хешу Perl предоставляет два варианта синтаксиса. В первом из них используется по одной инструкции присваивания для каждой пары ключ–значение:

```
$ cat hash1.pl
#!/usr/bin/perl -w

$hashvar1{boat} = "tuna";
$hashvar1{"number five"} = 5;
$hashvar1{4} = "fish";
@arrayhash1 = %hashvar1;
say "@arrayhash1";

$ ./hash1.pl
boat tuna 4 fish number five 5
```

Внутри инструкции присваивания ключ помещается в фигурные скобки слева от знака равенства, а значение помещается справа от этого знака. Как показано в пре-

дыдущем примере, ключи и значения могут быть либо числовыми, либо строковыми значениями. Если строковые ключи не содержат пробелов, их не нужно помещать в двойные кавычки. В этом примере также показано, что ключи и значения, содержащиеся в хеше, можно выводить на экран, присвоив хеш переменной массива, а затем дав команду на вывод этой переменной, заключив ее в кавычки.

В следующем примере показан еще один способ присваивания значений хешу и показано, как использовать функции keys и values для извлечения ключей и значений из хеша. После присваивания значений хешу %hash2 hash2.pl вызывает функцию keys с аргументом %hash2 и присваивает получившийся список ключей массиву @array_keys. Затем программа использует функцию values для присваивания значений массиву @array_values.

```
$ cat hash2.pl
#!/usr/bin/perl -w

%hash2 = (
    boat => "tuna",
    "number five" => 5,
    4 => "fish",
    );

@array_keys = keys(%hash2);
say "   Ключи: @array_keys";

@array_values = values(%hash2);
say "Значения: @array_values";

$ ./hash2.pl
   Ключи: boat 4 number five
Значения: tuna fish 5
```

Поскольку Perl автоматически помещает в кавычки одиночное слово, которое находится слева от оператора =>, ставить кавычки вокруг слова boat в третьей строке программы не нужно. Но удаление кавычек, поставленных вокруг слов number five, приведет к генерации ошибки, поскольку в строке содержится пробел.

Управляющие структуры

Инструкции управления ходом программы изменяют порядок выполнения инструкций в Perl-программе. В главе 10 были подробно рассмотрены управляющие структуры оболочки bash с приложением блок-схем порядка выполнения программ. Управляющие структуры Perl выполняют те же самые функции, что и их аналоги, имеющиеся в оболочке bash, хотя синтаксис этих двух языков различается. Описание каждой управляющей структуры в этом разделе перекликается с рассмотрением такой же управляющей структуры оболочки bash.

Слова, набранные в описании синтаксиса ***полужирным курсивом***, являются элементами, предоставляемыми структуре, чтобы получить от нее желаемый эффект; слова, набранные нежирным наклонным шрифтом, являются ключевыми словами Perl, которые используются для идентификации управляющей структуры, а группа

символов *{...}* представляет блок инструкций (см. раздел «Терминология» на с. 507). Многие из этих структур используют выражения, обозначенные как *выражения*, управляющие их выполнением. Пример и описание синтаксиса показаны ниже, в разделе «Управляющие структуры if и unless» на с. 521.

Управляющие структуры if и unless

Управляющие структуры if и unless являются составными инструкциями, имеющими следующий синтаксис:

```
if (выражение) {...}
unless (выражение) {...}
```

Эти структуры отличаются только смыслом проводимого ими теста. Структура if выполняет блок инструкций, *если* ***выражение*** вычисляется в *true*, а структура unless выполняет блок инструкций, *пока* ***выражение*** не будет вычислено в *true* (то есть если *выражение* вычисляется в *false*).

Слово *if* набрано нежирным шрифтом, потому что оно является ключевым словом и должно быть именно в том виде, в котором показано. А ***выражение*** является выражением, которое Perl вычисляет и выполняет блок (см. раздел «Терминология» на с. 507) инструкций, представленный группой символов *{...}*, если ***выражение*** вычисляется так, как этого требует управляющая структура.

Операторы проверки файлов

Выражение в следующем примере, `-r memo1`, использует оператор проверки файла `-r` для определения, существует ли файл memo1 в рабочем каталоге, а также имеется ли у пользователя право на его чтение. Хотя этот оператор проводит проверку только факта наличия у вас права на чтение, наличие такого права предполагает существование для вас этого файла, поэтому данный текст проводит также скрытую проверку присутствия файла в каталоге. (В Perl используются такие же операторы проверки файлов, что и в оболочке bash; см. табл. 10.1.) Если это *выражение* вычисляется в *true*, Perl выполняет блок инструкций (в данном случае этот блок состоит из одной инструкции), находящийся между фигурными скобками. Если выражение вычисляется в *false*, Perl пропускает блок инструкций. В любом случае, после этого Perl осуществляет выход из программы и возвращает управление оболочке.

```
$ cat if1.pl
#!/usr/bin/perl -w
if (-r "memo1") {
   say "Файл 'memo1' существует и может быть прочитан.";
   }

$ ./if1.pl
Файл 'memo1' существует и может быть прочитан.
```

Далее следует та же самая программа, написанная с использованием постфиксного синтаксиса if. Какой из двух синтаксисов следует применить, зависит от того, какая часть инструкции важнее для того, кто читает код.

```
$ cat if1a.pl
#!/usr/bin/perl -w
say "Файл 'memo1' существует и может быть прочитан." if (-r "memo1");
```

В следующем примере для вывода приглашения на стандартный вывод используется инструкция print, а для чтения строки из стандартного ввода и присваивания ее значения переменной $entry используется инструкция $entry = <>;. Чтение со стандартного ввода, работа с другими файлами и использование волшебного описателя файла (<>) для чтения файлов, указанных в командной строке, рассмотрены в разделе «Работа с файлами» на с. 530.

Операторы сравнения

Для сравнения чисел и строк в Perl используются различные операторы. В табл. 11.2 перечислены числовые и строковые операторы сравнения. В следующем примере в *выражении* инструкции if используется оператор сравнения чисел ==, который сравнивает значение, введенное пользователем, с числом 28. Этот оператор выполняет числовое сравнение, поэтому пользователь может ввести 28, 28.0 или 00028, и во всех случаях результат сравнения будет вычислен в *true*. Поскольку сравнение является числовым, Perl игнорирует как поставленные вокруг пробельные символы, так и разделитель строки, следующий за вводом пользователя. Ключ -w заставляет Perl выдавать предупреждение, если пользователь вводит нечисловое значение и программа использует это значение в арифметическом выражении; а без этого ключа Perl вычисляет выражение в *false*.

```
$ cat if2.pl
#!/usr/bin/perl -w
print "Введите 28: ";
$entry = <>;
if ($entry == 28) {                # использование == для сравнения чисел
    print "Спасибо за то, что вы ввели 28.\n";
    }
print "Конец.\n";

$ ./if2.pl
Введите 28: 28.0
Спасибо за то, что вы ввели 28.
Конец.
```

Таблица 11.2. Операторы сравнения

Числовые операторы	Строковые операторы	Значение, возвращенное на основе отношения значения, стоящего перед оператором, к значению, стоящему после него
==	eq	*True*, если равно
!=	ne	*True*, если не равно
<	lt	*True*, если меньше чем
>	gt	*True*, если больше чем
<=	le	*True*, если меньше чем или равно
>=	ge	*True*, если больше чем или равно
<=>	cmp	0 — если равно, 1 — если больше чем, –1 — если меньше чем

Следующая программа похожа на предыдущую, за исключением того, что она тестирует равенство двух строк. Функция chomp (см. раздел «chomp и chop» на с. 531) удаляет замыкающий разделитель строк из пользовательского ввода — без этой функции сравниваемая строка никогда не будет соответствовать той, с которой она сравнивается. Оператор сравнения eq проводит сравнение строк. В данном примере

результат сравнения строк будет равен *true*, если пользователь введет строку five. Лидирующие или замыкающие пробельные символы приведут к тому, что результат в строке 5 будет вычислен в *false*, хотя ни одна из этих записей не сгенерирует ошибку, поскольку они являются вполне допустимыми строками.

```
$ cat if2a.pl
#!/usr/bin/perl -w
print "Введите слово 'five': ";
$entry = <>;
chomp ($entry);
if ($entry eq "five") {           # использование eq для сравнения строк
   print "Спасибо за то, что ввели 'five'.\n";
   }
print "Конец.\n";

$ ./if2a.pl
Введите слово 'five': five
Спасибо за то, что ввели 'five'.
Конец.
```

Управляющая структура if...else

Управляющая структура if...else является составной инструкцией, похожей на имеющуюся в оболочке bash управляющую структуру if...then...else (см. раздел «Управляющая структура if...then...else» на с. 421). Ее применение позволяет реализовать разветвление выполнения программы на два направления. Для этой структуры используется следующий синтаксис:

*if (**выражение**) {...} else {...}*

die

В следующей программе пользователь получает приглашение на ввод двух разных чисел, которые сохраняются в переменных $num1 и $num2. Если пользователь вводит два одинаковых числа, структура if выполняет функцию die, которая отправляет свои аргументы на стандартную ошибку и прерывает выполнение программы.

Если пользователь вводит разные числа, структура if...else сообщает о том, какое из них больше. Поскольку *выражение* выполняет сравнение чисел, программа принимает числа, включающие десятичные точки.

```
$ cat ifelse.pl
#!/usr/bin/perl -w
print "Введите число: ";
$num1 = <>;
print "Введите еще одно число, отличное от первого: ";
$num2 = <>;

if ($num1 == $num2) {
   die ("Пожалуйста, введите два разных числа.\n");
   }
if ($num1 > $num2) {
   print "Первое число больше второго.\n";
   }
```

```
else {
    print "Первое число меньше второго.\n";
    }

$ ./ifelse.pl
Введите число: 8
Введите еще одно число, отличное от первого: 8
Пожалуйста, введите два разных числа.

$ ./ifelse.pl
Введите число: 5.5
Введите еще одно число, отличное от первого: 5
Первое число больше второго.
```

Управляющая структура if...elsif...else

Имеющаяся в Perl управляющая структура if...elsif...else похожа на управляющую структуру if...then...elif оболочки bash (см. раздел «Управляющая структура if...then...elif» на с. 423) и является составной инструкцией, реализующей вложенный набор структур if...else. Для этой структуры используется следующий синтаксис:

*if (**выражение**) {...} elsif {...} ... else {...}*

Следующая программа делает то же самое, что и предыдущая программа ifelse.pl, но уже с использованием структуры if...elsif...else. Инструкция die заменена инструкцией print, поскольку последняя инструкция программы отображает сообщение об ошибке, а после выполнения этой инструкции программа все равно заканчивает свою работу. Чтобы заставить Perl отправить это сообщение не на стандартный вывод, а на стандартную ошибку, можно воспользоваться описателем STDERR (см. раздел «Работа с файлами» на с. 530).

```
$ cat ifelsif.pl
#!/usr/bin/perl -w
print "Введите число: ";
$num1 = <>;
print "Введите еще одно число, отличное от первого: ";
$num2 = <>;

if ($num1 > $num2) {
        print "Первое число больше второго.\n";
        }
    elsif ($num1 < $num2) {
        print "Первое число меньше второго.\n";
        }
    else {
        print "Пожалуйста, введите два разных числа.\n";
        }
```

Управляющие структуры foreach и for

Ключевые слова Perl foreach и for являются синонимами, взаимозаменяемыми в любом контексте. Эти структуры являются составными инструкциями, имеющими два

варианта синтаксиса. Некоторые программисты используют один вариант синтаксиса с foreach, а другой — с for, хотя делать это совсем не обязательно. В данной книге для обоих вариантов используется ключевое слово foreach.

foreach: Синтаксис 1

Первый вариант синтаксиса для управляющей структуры foreach похож на синтаксис структуры for...in, которая используется в оболочке (см. раздел «Управляющая структура for...in» на с. 429):

foreach|for [переменная] (список) {...}

где *список* является списком выражений или переменных. Perl выполняет блок инструкций для каждого элемента списка, последовательно присваивая *переменной* значение одного элемента списка при каждой итерации, начиная с первого элемента. Если не указать *переменную*, Perl присвоит значение переменной `$_` (см. раздел «$_» на с. 529).

В следующей программе показана простая структура foreach. При первом проходе цикла Perl присваивает строку `Mo` переменной `$item`, а инструкция say выводит значение этой переменной, за которым следует разделитель строк. При втором и третьем проходах цикла переменная `$item` получает значения `Larry` и `Curly`. Когда в списке не останется ни одного элемента, Perl продолжит выполнение программы с инструкции, которая следует за структурой foreach. В данном случае программа завершает свою работу. Если эта программа жалуется на недоступность функции say, обратитесь к разделу «Использование свойства say (use feature 'say')» на с. 510.

```
$ cat foreach.pl
foreach $item ("Mo", "Larry", "Curly") {
    say "$item приветствует вас.";
    }

$ perl foreach.pl
Mo приветствует вас.
Larry приветствует вас.
Curly приветствует вас.
```

Используя переменную `$_`, эту программу можно написать следующим образом:

```
$ cat foreacha.pl
foreach ("Mo", "Larry", "Curly") {
    say "$_ приветствует вас.";
    }
```

В следующей программе используется массив:

```
$ cat foreachb.pl
@stooges = ("Mo", "Larry", "Curly");
foreach (@stooges) {
    say "$_ приветствует вас.";
    }
```

В следующей программе используется постфиксный синтаксис foreach:

```
$ cat foreachc.pl
@stooges = ("Mo", "Larry", "Curly");
say "$_ приветствует вас." foreach @stooges;
```

Переменная цикла (в предыдущих примерах — $item и $_) ссылается на элементы в *списке*, которые заключены в круглые скобки. При изменении *переменной* цикла изменяется и элемент *списка*. Функция uc возвращает переведенную в верхний регистр версию своих аргументов. В следующем примере показано, что изменение *переменной* цикла $stooge приводит к изменению массива @stooges:

```
$ cat foreachd.pl
@stooges = ("Mo", "Larry", "Curly");
foreach $stooge (@stooges) {
    $stooge = uc $stooge;
    say "$stooge приветствует вас.";
    }
say "$stooges[1] в верхнем регистре"

$ perl foreachd.pl
MO приветствует вас.
LARRY приветствует вас.
CURLY приветствует вас.
LARRY в верхнем регистре
```

Пример последовательного перебора аргументов командной строки показан далее в разделе «@ARGV» на с. 532.

Инструкции last и next

Имеющиеся в Perl инструкции last и next позволяют прервать цикл и являются аналогами имеющихся в оболочке Bourne Again Shell инструкций break и continue (см. раздел «Инструкции break и continue» на с. 438). Инструкция last передает управление той инструкции, которая следует за блоком инструкций, управляемым структурой цикла, завершая тем самым выполнение цикла. Инструкция next передает управление в конец блока инструкций, после чего выполнение цикла продолжается со следующей итерации.

В следующей программе структура if проверяет, не равно ли значение переменной $item строке two. Если равно, то структура выполняет следующую инструкцию, которая пропускает инструкцию say, а затем продолжает работу выполнением следующей итерации цикла. Если заменить инструкцию next инструкцией last, Perl осуществит выход из цикла и не выведет three. Если эта программа жалуется на недоступность функции say, обратитесь к разделу «Использование свойства say (use feature 'say')» на с. 510.

```
$ cat foreach1.pl
foreach $item ("one", "two", "three") {
    if ($item eq "two") {
        next;
        }
    say "$item";
    }

$ perl foreach1.pl
one
three
```

foreach: Синтаксис 2

Второй вариант синтаксиса структуры foreach похож на структуру for, которая используется в языке C:

foreach|for ***(выражение1; выражение2; выражение3)*** *{...}*

Выражение1 инициализирует цикл foreach; Perl вычисляет *выражение1* только один раз, перед выполнением блока инструкций. *Выражение2* является условием завершения цикла; Perl вычисляет его перед каждым проходом через блок инструкций и выполняет этот блок, если *выражение2* вычисляется в *true*. Perl вычисляет *выражение3* после каждого прохода через блок инструкций — обычно оно увеличивает на единицу значение переменной, являющейся частью *выражения2*.

В следующем примере программа foreach2.pl приглашает пользователя ввести три числа; выводит первое число; многократно увеличивает это число на значение второго числа, выводя каждый результат до тех пор, пока он не будет больше, чем третье число, после чего завершает работу. Волшебный описатель файла (<>) рассмотрен в разделе «Работа с файлами»на с. 530.

```
$ cat ./foreach2.pl
#!/usr/bin/perl -w

print "Введите начальное число: ";
$start = <>;

print "введите конечное число: ";
$end = <>;

print "Введите приращение: ";
$incr = <>;

if ($start >= $end || $incr < 1) {
    die ("Начальное число должно быть меньше конечного.\n",
    "и приращение должно быть больше нуля.\n");
    }
foreach ($count = $start+0; $count <= $end; $count += $incr) {
    say "$count";
    }

$ ./foreach2.pl
Введите начальное чило: 2
введите конечное число: 10
Введите приращение: 3
2
5
8
```

После вывода приглашения на ввод трех чисел предыдущая программа проверяет, не окажется ли начальное число больше или равно конечному, или не окажется ли приращение меньше 1. Символы || являются булевым оператором ИЛИ (OR); выражение в круглых скобках, которое следует за if, вычисляется в *true*, если либо выражение до, либо выражение после этого оператора вычисляется в *true*.

Инструкция foreach начинается с присваивания значения `$start+0` переменной `$count`. Прибавление 0 (нуля) к строке `$start` заставляет Perl работать в числовом контексте, удаляя замыкающий разделитель строк при выполнении присваивания. Без этой поправки программа будет выводить лишний разделитель строк, следующий за первым выводимым числом.

Управляющие структуры while и until

Управляющие структуры while (см. главу 10, раздел «Управляющая структура while») и until (см. раздел «Управляющая структура until» на с. 436) являются составными инструкциями, реализующими условные циклы. Для этих структур используется следующий синтаксис:

while ***(выражение)*** *{...}*
until ***(выражение)*** *{...}*

Эти структуры отличаются только смыслом условия их завершения. Структура while многократно выполняет блок инструкций, *пока* ***выражение*** вычисляется в *true*, а структура until продолжает его выполнение *до тех пор, пока* ***выражение*** не будет вычислено в *true* (то есть до тех пор, пока оно вычисляется в *false*).

В следующем примере показана технология, применяющаяся для чтения и обработки ввода до тех пор, пока он не закончится. Хотя этот пример показывает ввод, поступающий от пользователя (стандартный ввод), технология точно также работает и для ввода, поступающего из файла (см. пример в разделе «chomp и chop»). Чтобы подать сигнал о конце файла, пользователь самостоятельно нажимает комбинацию CONTROL+D.

В этом примере используется *выражение* `$line = <>`. В этом выражении для чтения одной строки из стандартного ввода и присваивания прочитанной строки переменной `$line` используется волшебный описатель файла (<>; см. раздел «Работа с файлами» на с. 530). Эта инструкция вычисляется в *true* до тех пор, пока она считывает данные. Как только она достигнет конца файла, инструкция вычисляется в *false*. Цикл while продолжает выполнение блока инструкций (в данном примере он состоит только из одной инструкции), пока есть данные для чтения.

```
$ cat while1.pl
#!/usr/bin/perl -w
$count = 0;
while ($line = <>) {
    print ++$count, ". $line";
    }
print "\n Введено строк: $count.\n";

$ ./while1.pl
Доброе утро.
1. Доброе утро.
Сегодня понедельник.
2. Сегодня понедельник.
CONTROL+D
Введено строк: 2.
```

В предыдущем примере в переменной $count отслеживается количество строк, введенных пользователем. За счет помещения оператора инкремента ++ перед именем переменной (++$count; здесь используется так называемый оператор преинкремента) значение переменной увеличивается на единицу, перед тем как Perl произведет вычисление. Вместо этого можно было бы инициализировать переменную $count значением 1, и увеличить ее значение с помощью выражения $count++ (то есть использовать постинкремент), но тогда в последней инструкции print переменная $count имела бы значение на единицу большее, чем количество введенных строк.

$. В переменной $. сохраняется количество строк, считанных программой из ввода. Используя переменную $., можно переписать предыдущий пример следующим образом:

```
$ cat while1a.pl
#!/usr/bin/perl -w
while ($line = <>) {
        print $., ". $line";
        }
print "\nВведено строк: $..\n";
```

$_ Код Perl зачастую удается упростить за счет применения переменной $_. Эту переменную можно использовать в программе на Perl во многих местах. О переменной $_ нужно думать как об объекте, с которым производятся действия. Она является операндом для многих операций. К примеру, в следующем фрагменте кода переменная $line используется для обработки строки. В этом фрагменте строка считывается в переменную $line, из нее с помощью инструкции chomp удаляются все замыкающие разделители строк (см. раздел «chomp и chop» на с. 531), и проводится проверка соответствия содержимого переменной $line регулярному выражению.

```
while (my $line = <>) {
    chomp $line;
    if ($line =~ /regex/) …
}
```

Чтобы избавиться от переменной $line за счет использования переменной $_, можно переписать этот код следующим образом:

```
while (my $_ = <>) {
    chomp $_;
    if ($_ =~ /regex/) …
}
```

Поскольку переменная $_ является операндом по умолчанию в каждом из этих примеров, переменную $_ можно вообще опустить:

```
while (<>) {            # Чтение в переменную $_
    chomp;              # Обработка переменной $_ с помощью инструкции chomp
    if (/regex/) …      # Если значение переменной $_ соответствует регулярному
                        # выражению regex
}
```

Работа с файлами

Открытие файла и присваивание описателя

Описатель — это имя, которое можно использовать в программе на Perl для ссылки на файл или процесс, открытый для чтения и (или) записи. При работе с оболочкой эти описатели известны нам как описатели файлов (см. раздел «Описатели файлов» на с. 449). В процессе работы с оболочкой, перед тем как запустить программу автоматически, ядро открывает описатели для стандартного ввода и стандартного вывода (см. раздел «Стандартный ввод и стандартный вывод» на с. 143), а также для стандартной ошибки (см. раздел «Перенаправление стандартной ошибки» на с. 288). Когда программа завершает свою работу, ядро эти описатели закрывает. У этих описателей соответственно имеются имена STDIN, STDOUT и STDERR. Чтобы прочитать данные из других файлов или процессов или записать в них данные, нужно открыть описатели самостоятельно. Инструкция открытия open имеет следующий синтаксис

*open (**описатель_файла**, ['**режим**',] "**ссылка_на_файл**");*

где *описатель_файла* — это имя описателя или переменная, которая будет использоваться в программе для ссылки на файл или процесс, указанный *ссылкой_на_файл*. Если опустить *режим* или указать *режим* <, Perl открывает файл для ввода (чтения). Указание *режима* > используется для усечения файла и записи в него данных, а указание *режима* >> — для добавления данных к файлу. Чтение из процессов и запись в них рассмотрены в разделе «Прием вывода от процесса» на с. 546.

Запись в файл

Функция print записывает вывод в файл или процесс. Для инструкции print используется следующий синтаксис:

*print [**описатель_файла**] "**текст**";*

где *описатель_файла* — это имя описателя, указанное в инструкции open, а *текст* — это информация, которую нужно вывести. В качестве *описателя_файла*, как ранее объяснялось, можно также указать STDOUT или STDERR. За исключением того случая, когда информация отправляется на стандартный вывод, в инструкции print нужно указывать описатель. Ставить запятую после *описателя_файла* не нужно. Также не нужно заключать аргументы инструкции print в круглые скобки, поскольку это может вызвать ряд проблем.

Чтение из файла

Следующее выражение используется для чтения одной строки, включая разделитель строк (\n), из файла или процесса с помощью *описателя_файла*:

<описатель_файла>

Это выражение обычно используется в такой инструкции, как

```
$line = <IN>;
```

с помощью которой производится чтение в переменную $line одной строки из файла или процесса, идентифицируемого с помощью описателя IN.

Волшебный описатель файла (<>)

Чтобы облегчить чтение из файлов, указанных в командной строке, или из стандартного ввода, Perl предоставляет волшебный описатель файла. В данной книге этот описатель используется в большинстве примеров. Вместо предыдущей строки можно использовать следующее выражение:

```
$line = <>;
```

Этот описатель файла заставляет программу на Perl работать наподобие многих утилит Linux: она осуществляет чтение из стандартного ввода до тех пор, пока не будет вызвана с одним или несколькими аргументами, тогда она будет читать данные из файлов, указанных в качестве аргументов. О том, как это свойство работает с командой cat, объяснено в разделе «Утилиты, воспринимающие ввод из файла или со стандартного устройства ввода» на с. 148.

В следующем примере инструкция print, находящаяся в первой строке, включает необязательный описатель STDOUT; в следующей инструкции print этот описатель опущен; в последней инструкции print используется описатель файла STDERR, который заставляет вывести вывод на стандартную ошибку.

Первая инструкция print приглашает пользователя что-нибудь ввести. Строка, выводимая этой инструкцией, заканчивается не разделителем строк, а пробелом, чтобы пользователь мог ввести информацию в ту же самую строку приглашения. Затем во второй строке используется волшебный описатель, чтобы прочитать из стандартного ввода одну строку, которая присваивается переменной $userline. Благодаря использованию волшебного описателя, если вызвать программу file1.pl с аргументом, являющимся именем файла, она считывает одну строку не из стандартного ввода, а из файла с этим именем. Командная строка, запускающая программу file1.pl, использует символы 2> (см. раздел «Описатели файлов» на с. 289) для перенаправления стандартной ошибки (вывода третьей инструкции print) в файл file1.err.

```
$ cat file1.pl
print STDOUT "Введите что-нибудь: ";
$userline = <>;
print "1>>>$userline<<<\n";
chomp ($userline);
print "2>>>$userline<<<\n";
print STDERR "3. Сообщение  об ошибке.\n";

$ perl file1.pl 2> file1.err
Введите что-нибудь: привет
1>>>привет
<<<
2>>>привет<<<

$ cat file1.err
3. Сообщение об ошибке.
```

chomp и chop Две инструкции print, следующие за пользовательским вводом в программе file1.pl, сразу же выводят значение переменной $userline, перед которым следуют символы «больше» (>), а сразу после него — символы «меньше» (<). Первая из трех этих инструкций показывает, что переменная $userline включает в себя разделитель строк: символы «меньше», следующие за введенной пользователем строкой, появляется на следующей строке. Функция chomp удаляет из строки замыкающий разделитель строк, если таковой в ней присутствует. После обработки переменной $userline с помощью функции chomp инструкция print показывает, что это значение больше не содержит разделитель строк. (Функция chop работает так же, как функция chomp, за исключением того, что она удаляет из строки любой замыкающий символ.)

В следующем примере показано, как осуществляется чтение из файла. Для назначения лексического файлового описателя `$infile` файлу /usr/share/dict/words в нем используется инструкция open. При следующей итерации инструкции while выражение читает строку из файла, представленного описателем `$infile`, и присваивает строку переменной `$line`. Когда инструкция while доберется до конца файла, выражение будет вычислено в false, и тогда управление будет передано за пределы структуры while. Блок, состоящий из одной инструкции, выводит строку в том виде, в котором она была прочитана из файла, включая разделитель строк.

Эта программа копирует файл /usr/share/dict/words на стандартный вывод. Затем для отправки вывода утилите head (см. раздел «head: Отображение начальных строк файла» на с. 72), которая выводит первые четыре строки файла (первая строка — пустая), используется канал (|, см. раздел «| (канал): Организация связи между процессами» на с. 76).

```
$ cat file2.pl
open (my $infile, "/usr/share/dict/words") or die "Не могу открыть словарь: $!\n";
while ($line = <$infile>) {
    print $line;
    }
$ perl file2.pl | head -4
A
A's
AOL
```

$! В переменной `$!` содержится последняя системная ошибка. В числовом контексте в ней содержится номер системной ошибки, а в строковом — строка системной ошибки. Если файл words отсутствует в системе, программа file2.pl выводит следующее сообщение:

```
Не могу открыть словарь: No such file or directory
                         такой файл или каталог отсутствует
```

Если у вас нет права на чтение файла, программа выводит следующее сообщение:

```
Не могу открыть словарь: Permission denied
                         Отсутствие прав доступа
```

Вывод значения переменной `$!` предоставляет пользователю больше информации о причинах отказа, чем простое сообщение о том, что программа не может открыть файл.

ПРИ ОТКРЫТИИ ФАЙЛА НУЖНО ВСЕГДА ПРОВЕРЯТЬ, НЕ ПРОИЗОШЛА ЛИ ОШИБКА

Когда программа на Perl безуспешно пытается открыть файл, она не выводит сообщение об ошибке до тех пор, пока не проверит, что инструкция open вернула ошибку. В файле file2.pl оператор or в инструкции open заставляет Perl выполнить инструкцию die (см. раздел «die»на с. 523), если инструкция open не выполнится. Инструкция die отправляет на стандартную ошибку сообщение «Не могу открыть словарь», за которым следует строка системной ошибки, и завершает выполнение программы.

@ARGV В массиве `@ARGV` содержатся аргументы командной строки, с которыми был вызван Perl. Когда следующая программа вызывается со списком имен файлов, она выводит первую строку каждого файла. Если программа не может прочитать файл, инструкция die отправляет сообщение об ошибке на стандартную

ошибку и завершает работу программы. Структура foreach осуществляет последовательный перебор аргументов командной строки, представленных в переменной @ARGV, присваивая по очереди значение каждого аргумента переменной $filename. Блок структуры foreach начинается с инструкции open. Perl выполняет инструкцию open, которая предшествует булеву оператору ИЛИ (or), или, если эта инструкция потерпит неудачу, Perl выполняет инструкцию, которая следует за оператором or (то есть инструкцию die). В результате Perl либо открывает файл по имени, которое содержится в переменной $filename, и назначает IN его описателем, либо (если не может открыть файл) выполняет инструкцию die и завершает выполнение программы. Инструкция print выводит имя файла, за которым следует двоеточие, и первую строку этого файла. Когда Perl принимает выражение $line = <IN> в качестве аргумента инструкции print, он выводит значение переменной $line после операции присваивания. После чтения строки из файла программа закрывает этот файл.

```
$ cat file3.pl
foreach $filename (@ARGV) {
    open (IN, $filename) or die "Не могу открыть файл '$filename': $!\n";
    print "$filename: ", $line = <IN>;
    close (IN);
    }

$ perl file3.pl f1 f2 f3 f4
f1: First line of file f1.
f2: First line of file f2.
Не могу открыть файл 'f3': No such file or directory
                        Такой файл или каталог отсутствует
```

Следующий пример похож на предыдущий, за исключением того, что в нем используются свойства Perl, упрощающие код. В этом примере, когда не получается открыть файл, выхода из программы не происходит. Вместо этого Perl выводит сообщение об ошибке и продолжает выполнение программы. В первой строке программы для объявления переменной $filename в качестве лексической используется ключевое слово my. Затем в структуре while для открытия и чтения каждой строки каждого файла, указанного в аргументах командной строки, используется волшебный описатель файла; имя файла содержится в переменной $ARGV. Когда файлов для чтения больше не останется, условие [(<>)]структуры while вычисляется в *false*, while передает управление за пределы своего блока, и программа завершает свою работу. Perl берет на себя все операции открытия и закрытия файлов, и вам не нужно вводить код, выполняющий эти задачи. Perl также выполняет проверку на возникновение ошибок.

Программа выводит первую строку каждого файла, чье имя указано в аргументе командной строки. При каждом проходе блока структуры while считывается новая срока. Когда будет закончена работа с одним файлом, начинается чтение из другого файла. Внутри блока while инструкция if проверяет, не обрабатывает ли он новый файл. Если он обрабатывает новый файл, то блок инструкции if выводит имя этого файла и первую его строку, а затем присваивает переменной $filename имя нового файла ($ARGV).

```
$ cat file3a.pl
my $filename;
while (<>) {
    if ($ARGV ne $filename) {
```

```
        print "$ARGV: $_";
        $filename = $ARGV;
    }
}
```

```
$ perl file3a.pl f1 f2 f3 f4
f1: First line of file f1.
f2: First line of file f2.
Can't open f3: No such file or directory at file3a.pl line 3, <> line 3.
Не могу открыть f3: Такой файл или каталог отсутствует; file3a.pl строка 3,
<> строка 3.
f4: First line of file f4.
```

Сортировка

reverse

Функция sort возвращает на основе локальных установок среды элементы массива, выстроенные в алфавитном или числовом порядке. Функция reverse не имеет отношения к функции sort; она просто возвращает элементы массива в обратном порядке.

В первых двух строках следующей программы элементам массива @colors присваиваются значения, которые затем выводятся на экран. В каждой из следующих пар строк используется функция sort, которая размещает значения из массива @colors в определенном порядке, присваивает результат массиву @scolors и выводит содержимое @scolors на экран. При этой сортировке буквы в верхнем регистре идут раньше, чем в нижнем. Обратите внимание в отсортированном выводе на позицию элементов Orange и Violet, каждый из которых начинается с букв в верхнем регистре. Первая инструкция присваивания в первых двух парах строк использует полный синтаксис функции sort, включая блок {$a cmp $b}, предписывающий Perl использование подпрограммы cmp, которая проводит сравнение строк и размещает результат в возрастающем порядке. Когда в инструкции sort опускается блок кода, как в случае со второй инструкцией присваивания, Perl также осуществляет текстовую сортировку в возрастающем порядке.

```
$ cat sort3.pl
@colors = ("red", "Orange", "yellow", "green", "blue", "indigo", "Violet");

say "@colors";

@scolors = sort {$a cmp $b} @colors;          # сортировка в возрастающем порядке
say "@scolors";                               # с явно указанным блоком

@scolors = sort @colors;                      # сортировка в возрастающем порядке
say "@scolors";                               # с подразумеваемым блоком

@scolors = sort {$b cmp $a} @colors;          # сортировка в убывающем порядке
say "@scolors";

@scolors = sort {lc($a) cmp lc($b)} @colors;  # сортировка в возрастающем
say "@scolors";                               # порядке без учета регистра
```

```
$ perl sort3.pl
red Orange yellow green blue indigo Violet
Orange Violet blue green indigo red yellow
Orange Violet blue green indigo red yellow
yellow red indigo green blue Violet Orange
blue green indigo Orange red Violet yellow
```

В третей сортировке предыдущего примера используется обратный порядок расстановки переменных `$a` и `$b` в блоке, указывающем на сортировку по убыванию. В последней сортировке проводится приведение строк перед сравнением в нижний регистр, тем самым производится такая сортировка, при которой все буквы верхнего регистра рассматриваются как буквы нижнего регистра. В результате `Orange` и `Violet` сортируются в алфавитном порядке.

Чтобы провести числовую сортировку, вместо подпрограммы cmp нужно применить подпрограмму <=>. В следующем примере показана числовая сортировка в возрастающем и убывающем порядке:

```
$ cat sort4.pl
@numbers = (22, 188, 44, 2, 12);

print "@numbers\n";

@snumbers = sort {$a <=> $b} @numbers;
print "@snumbers\n";

@snumbers = sort {$b <=> $a} @numbers;
print "@snumbers\n";

$ perl sort4.pl
22 188 44 2 12
2 12 22 44 188
188 44 22 12 2
```

Подпрограммы

Все переменные являются пакетными (см. раздел «Терминология» на с. 507), пока для определения их в качестве лексических переменных (см. раздел «Терминология») не будет использована функция my. Лексические переменные, определенные в подпрограмме, являются локальными по отношению к этой подпрограмме.

Следующая программа включает основную часть и подпрограмму по имени add(). Эта программа использует переменные по имени `$one`, `$two` и `$ans`, и все они являются пакетными переменными: они доступны как основной программе, так и подпрограмме. При вызове подпрограммы ей не передаются никакие значения, а при выходе из нее она не возвращает никаких значений. Подобная ситуация не является типовой, она просто демонстрирует, что все переменные являются пакетными до тех пор, пока для их объявления лексическими не будет использована функция my.

Программа subroutine1.pl присваивает значения двум переменным и вызывает подпрограмму, которая складывает значения двух переменных и присваивает результат еще одной переменной. Основная часть программы выводит результат.

```
$ cat subroutine1.pl
$one = 1;
$two = 2;
add();
print "Ответ: $ans\n";

sub add {
    $ans =$one + $two
    }

$ perl subroutine1.pl
Ответ: 3
```

Следующий пример похож на предыдущий, но в отличие от него в подпрограмме используется инструкция return, возвращающая значение в основную программу. Программа присваивает возвращенное значение переменной $ans и выводит это значение. В этом случае снова используются только пакетные переменные.

```
$ cat subroutine2.pl
$one = 1;
$two = 2;
$ans = add();
print "Ответ: $ans\n";

sub add {
    return ($one + $two)
    }

$ perl subroutine2.pl
Ответ: 3
```

Сохранение локальности переменных в подпрограмме очень важно во многих случаях. Подпрограмма в следующем примере изменяет значение переменных и изолирует вызывающую программу от этих изменений за счет объявления и использования лексических переменных. Этот случай более типичен, чем предыдущий.

@_

Когда при вызове подпрограммы ей передаются какие-нибудь значения, Perl обеспечивает доступность этих значений в подпрограмме через массив по имени @_. Хотя массив @_ является локальным по отношению к подпрограмме, его элементы являются псевдонимами тех параметров, с которыми была вызвана подпрограмма. Изменение значения в массиве @_ приводит к изменению значения исходной переменной, что может быть весьма нежелательным явлением. В следующей программе этой проблемы удается избежать путем присваивания переданных подпрограмме значений лексическим переменным.

Программа subroutine3.pl вызывает подпрограмму addplusone() с двумя переменными в качестве аргументов и присваивает переменной значение, возвращенное подпрограммой. Первая инструкция подпрограммы объявляет две лексические переменные и присваивает им значения из массива @_. Функция my объявляет эти переменные лексическими. Хотя функцию my можно использовать для объявления переменных и без присваивания им значений, чаще всего используется синтаксис, примененный в данном примере. Следующие две инструкции увеличивают значения лексических

переменных $lcl_one и $lcl_two на единицу. Имеющаяся в подпрограмме инструкция print выводит значение переменной $lcl_one. А инструкция return возвращает сумму двух значений лексических переменных, увеличенных на единицу.

```
$ cat subroutine3.pl
$one = 1;
$two = 2;
$ans = addplusone($one, $two);
print "Ответ: $ans\n";
print "Значение 'lcl_one' в основной программе: $lcl_one\n";
print "Значение 'one' в основной программе: $one\n";

sub addplusone {
    my ($lcl_one, $lcl_two) = @_;
    $lcl_one++;
    $lcl_two++;
    print "Значение 'lcl_one' в подпрограмме: $lcl_one\n";
    return ($lcl_one + $lcl_two)
    }

$ perl subroutine3.pl
Значение 'lcl_one' в подпрограмме: 2
Ответ: 5
Значение 'lcl_one' в основной программе:
Значение 'one' в основной программе: 1
```

После вывода результата, возвращенного подпрограммой, инструкции print в основной программе показывают, что переменная $lcl_one в основной программе не определена (она является локальной переменной подпрограммы), а значение переменной $one не изменилось.

В следующем примере демонстрируется еще один способ работы с параметрами, переданными подпрограмме. Эта подпрограмма не использует переменных, кроме тех, которые ей были переданы с помощью массива @_, и не изменяет значения никаких элементов этого массива.

```
$ cat subroutine4.pl
$one = 1;
$two = 2;
$ans = addplusone($one, $two);
print "Ответ: $ans\n";
sub addplusone {
    return ($_[0] + $_[1] + 2);
    }

$ perl subroutine4.pl
Ответ: 5
```

Последний пример этого раздела представляет более типичную подпрограмму Perl. Подпрограмма max() может быть вызвана с любым количеством числовых параметров и используется для возвращения значения наибольшего из аргументов. Для присваивания переменной $biggest первого аргумента, с которым была вызвана подпрограмма, и для смещения всех остальных аргументов в ней используется функция shift. После

использования функции shift аргумент номер 2 становится аргументом номер 1 (8), аргумент номер 3 становится аргументом номер 2 (64), а аргумент номер 4 становится аргументом номер 3 (2). Затем с помощью структуры foreach осуществляет последовательный перебор всех остальных аргументов (@_). При каждом проходе блока структуры foreach Perl присваивает переменной $_ очередное значение каждого аргумента. Переменной $biggest присваивается значение переменной $_, если оно больше, чем значение переменной $biggest. Когда подпрограмма max() завершит последовательный перебор своих аргументов, переменная $biggest будет содержать наибольшее значение, которое и будет возвращено этой подпрограммой.

```
$ cat subroutine5.pl
$ans = max (16, 8, 64, 2);
print "Максимальное значение: $ans\n";

sub max {
    my $biggest = shift;  # Присваивание первого и смещение всех остальных
                          # аргументов, переданных max()
    foreach (@_) {        # Последовательный перебор всех остальных аргументов
    $biggest = $_ if $_ > $biggest;
    }
return ($biggest);
}

$ perl subroutine5.pl
Максимальное значение: 64
```

Регулярные выражения

Регулярные выражения можно использовать во многих утилитах Linux, и практически все эти выражения за редким исключением применимы и к программам на Perl. Вдобавок ко всему Perl предлагает функции для работы с регулярными выражениями, которые позволяют осуществлять более сложную обработку строк. В данном разделе дается обзор регулярных выражений и описываются некоторые дополнительные функции регулярных выражений, доступные в Perl. Здесь также представлен синтаксис Perl, используемый для работы с регулярными выражениями.

Синтаксис и оператор =~

Ключ -l

Используемый в Perl ключ -l заставляет применять инструкции chomp к каждой строке ввода и устанавливать символы \n в конце каждой строки вывода. Примеры в этом разделе используют ключи Perl -l и -e (см. раздел «Запуск программы на Perl» на с. 508). Поскольку программа должна быть указана в качестве единого аргумента, имена программ на Perl заключены в примерах в одинарные кавычки. Оболочка понимает назначение кавычек и не передает их интерпретатору Perl.

Символ / как ограничитель по умолчанию

По умолчанию регулярные выражения в Perl ограничены слэшами (/). В первой программе для поиска шаблона ge в строке aged используется оператор =~. Смысл этого оператора можно выразить словом

«включает». Иначе говоря, оператор =~ определяет, соответствует ли регулярное выражение ge части строки aged. Регулярное выражение, используемое в этом примере, не содержит специальных символов; строка ge является частью строки aged. Поэтому выражение, заключенное в круглые скобки, вычисляется в *true*, и Perl выполняет инструкцию print.

```
$ perl -le 'if ("aged" =~ /ge/) {print "true";}'
true
```

Того же эффекта можно добиться путем использования постфиксной инструкции if:

```
$ perl -le 'print "true" if "aged" =~ /ge/'
true
```

ИСПОЛЬЗОВАНИЕ ДРУГИХ КЛЮЧЕЙ ВМЕСТЕ С КЛЮЧОМ -E

Когда с ключом -e используется другой ключ, программа должна следовать в командной строке сразу же за ключом -e. Как и многие другие утилиты, Perl позволяет сочетать ключи, следующие за одиночным дефисом; если ключ -e является одним из ключей, входящих в комбинацию, то в списке ключей он должен стоять последним. То есть можно использовать сочетание perl -l -e или perl -le, но ни в коем случае не сочетание perl -e -l или perl -el.

!~ Оператор !~ по своему смыслу является противоположностью оператора =~. Выражение в следующем примере вычисляется в *true*, потому что регулярное выражение xy не соответствует ни одной из частей строки aged:

```
$ perl -le 'print "true" if ("aged" !~ /xy/)'
true
```

Точка в составе регулярного выражения соответствует любому одному символу, поэтому регулярное выражение a..d соответствует строке aged:

```
$ perl -le 'print "true" if ("aged" =~ /a..d/)'
true
```

Для хранения регулярного выражения можно воспользоваться переменной. В следующем выражении *строка* приводится в качестве регулярного выражения:

qr/строка/

В следующем примере этот синтаксис используется для присваивания регулярного выражения /a..d/ (включая ограничители) переменной $re с последующим использованием этой переменной в качестве регулярного выражения:

```
$ perl -le '$re = qr/a..d/; print "true" if ("aged" =~ $re)'
true
```

Если нужно включить ограничитель в состав регулярного выражения, действие этого ограничителя нужно отключить. В следующем примере ограничитель по умолчанию, знак слэш (/), появляется в составе регулярного выражения. Чтобы Perl не интерпретировал символ / в строке /usr как конец регулярного выражения, тот символ /, который является частью регулярного выражения, отключен (процитирован) путем установки перед ним символа обратного слэша (\).

```
$ perl -le 'print "true" if ("/usr/doc" =~ /\/usr/)'
true
```

Цитирование нескольких символов путем установки перед каждым из них символа обратного слэша может усложнить чтение составного регулярного выражения.

Вместо этого можно поставить перед ограниченным регулярным выражением символ m и использовать в качестве ограничителей парный набор символов, например {}. В следующем примере знак вставки привязывает регулярное выражение к началу строки:

```
$ perl -le 'print "true" if ("/usr/doc" =~ m{^/usr})'
true
```

Такой же синтаксис можно использовать при присваивании регулярного выражения переменной:

```
$ perl -le '$pn = qr{^/usr}; print "true" if ("/usr/doc" =~ $pn)'
true
```

Замена строки и присваивание

Для подстановки строки (замены одной строки другой строкой), соответствующей регулярному выражению, в Perl используется синтаксис, показанный в следующем примере. Этот синтаксис аналогичен тому, который встречается в редакторах vim и sed. Во второй строке примера символ s перед регулярным выражением заставляет Perl осуществить подстановку строки, указанной между вторым и третьим слэшами (worst, строка замены), вместо той строки, которая соответствует регулярному выражению, указанному между первыми двумя слэшами (best). В этом синтаксисе подразумевается, что подстановка осуществляется в строке, хранящейся в переменной, стоящей слева от оператора =~.

```
$ cat re10a.pl
$stg = "This is the best!";
$stg =~ s/best/worst/;
print "$stg\n";

$ perl re10a.pl
This is the worst!
```

В табл. 11.3 перечислены некоторые так называемые метасимволы, которые в регулярных выражениях Perl рассматриваются в качестве специальных. Дополнительные сведения можно получить с помощью ввода команды `perldoc perlre`.

Таблица 11.3. Некоторые метасимволы регулярных выражений Perl

Символ	Соответствует
^ (знак вставки)	Привязке регулярного выражения к началу строки
$ (знак доллара)	Привязке регулярного выражения к концу строки
(...)	Заключению регулярного выражения в круглые скобки (см. раздел «Заключение выражений в круглые скобки»)
. (точка)	Любому одному символу за исключением разделителя строк (\n)
\\	Обратному слэшу (\)
\b	Границе слова (соответствие нулевой ширине)
\B	Границе, не являющейся границей слова ([^\b])
\d	Одной десятичной цифре ([0-9])
\D	Одному символу, не являющемуся десятичной цифрой ([^0-9] или [^\d])
\s (нижний регистр)	Одному пробельному символу — пробелу, разделителю строк, символу ввода, табуляции, перевода страницы

Символ	Соответствует
\S (верхний регистр)	Одному непробельному символу ([^\s])
\w (нижний регистр)	Одному символу слова (букве или цифре; [a-zA-Z0-9])
\W (верхний регистр)	Одному символу, не использующемуся в словах ([^\w])

Жадное соответствие

По умолчанию Perl ищет жадное соответствие, означающее, что регулярное выражение соответствует самой длинной из всех возможных строк. В следующем примере регулярное выражение /{.*} / соответствует открывающей фигурной скобке, за которой следует любая строка символов, закрывающей фигурной скобке и пробелу (то есть строке {это удали} может иметь две {это сохрани}). Perl подставляет вместо этого соответствия нулевую строку (//).

```
$ cat 5ha.pl
$string = "Строка {это удали} может иметь две {это сохрани} пары фигурных скобок.";
$string =~ s/{.*} //;
print "$string\n";

$ perl 5ha.pl
Строка пары фигурных скобок.
```

Нежадные соответствия

В следующем примере показывается классический способ поиска соответствия самой короткой строке, заключенной в фигурные скобки, из предыдущего примера. Этот тип соответствия называется нежадным или скупым соответствием. Имеющееся в нем регулярное выражение соответствует

- открывающей фигурной скобке, за которой следует
- символ, принадлежащий классу символов, в который включены все символы за исключением закрывающей фигурной скобки ([^}]), за которым следует
- нулевое или любое количество экземпляров предыдущего символа (*), за которым следует
- закрывающая фигурная скобка, за которой следует
- пробел.

(Знак вставки, являющийся первым символом класса символов, определяет класс, состоящий из всех символов, не соответствующих следующим за ним символам, поэтому [^}] соответствует любому символу, не являющемуся закрывающей фигурной скобкой.)

```
$ cat re5b.pl
$string = "Строка {это удали} может иметь две {это сохрани} пары фигурных скобок.";
$string =~ s/{[^}]*} //;
print "$string\n";

$ perl re5b.pl
Строка может иметь две {это сохрани} пары фигурных скобок.
```

Perl предоставляет сокращенную форму записи, позволяющую определить нежадное соответствие. В следующем примере вопросительный знак в группе символов {.*?}

заставляет регулярное выражение соответствовать самой короткой строке, которая начинается с открывающей фигурной скобки, за которой следует любая строка символов, за которой, в свою очередь, следует закрывающая фигурная скобка.

```
$ cat re5c.pl
$string = "Строка {это удали} может иметь две {это сохрани} пары фигурных скобок.";
$string =~ s/{.*?} //;
print "$string\n";

$ perl re5c.pl
Строка может иметь две {это сохрани} пары фигурных скобок.
```

Заключение выражений в круглые скобки

Части регулярного выражения можно заключить в скобки и повторно вызвать эти части в строке замены. Многие утилиты Linux используют для заключения регулярного выражения в скобки процитированные круглые скобки (то есть \(и \)). В регулярных выражениях Perl круглые скобки являются специальными символами. Perl опускает символы обратного слэша и использует для заключения в скобки регулярного выражения непроцитированные круглые скобки. А чтобы указать круглую скобку как обычный символ в составе регулярного выражения Perl, ее нужно процитировать.

В следующем примере для заключения в скобки части регулярного выражения используются непроцитированные круглые скобки. Затем та часть строки, которой соответствует заключенное в кавычки выражение, присваивается переменной, хранившей строку, в которой Perl первоначально вел поиск соответствия регулярному выражению.

Сначала программа присваивает строку `My name is Sam` переменной `$stg`. Следующая инструкция ищет соответствие регулярному выражению `/My name is (.*)/` в строке, хранящейся в переменной `$stg`. Часть регулярного выражения, заключенная в круглые скобки, соответствует строке `Sam`; переменная `$1` в строке замены соответствует первой (и в данном случае единственной) заключенной в скобки части регулярного выражения. В результате строка, хранящаяся в переменной `$stg`, заменяется строкой **Sam**.

```
$ cat re11.pl
$stg = "My name is Sam";
$stg =~ s/My name is (.*)/$1/;
print "Matched: $stg\n";

$ perl re11.pl
Matched: Sam
```

В следующем примере регулярное выражение используется для разбора строки на числа. Две переменные инициализируются для хранения строки, содержащей два числа. В третьей строке программы используется регулярное выражение для выделения первого числа в строке. Выражение `\D*` соответствует строке из нуля и более символов, не включающей цифры: специальный символ `\D` соответствует любому одному нецифровому символу. Замыкающий символ звездочки заставляет эту часть регулярного выражения искать жадное соответствие, не включающее цифру (оно соответствует строке `What is`). Заключенная в скобки часть регулярного выражения `\d+` соответствует строке из одной или нескольких цифр. Круглые скобки не влияют на

то, чему соответствует регулярное выражение; они позволяют переменной $1 в строке замены соответствовать тому, что соответствует заключенной в скобки части регулярного выражения. Заключительная часть регулярного выражения .* соответствует всей остальной строке. Эта строка кода присваивает значение первого числа в строке переменной $string.

Следующая строка аналогична предыдущей, но она присваивает значение второго числа в строке переменной $string2. Инструкция print выводит числа и результат вычитания второго числа из первого.

```
$ cat re8.pl
$string = "Сколько будет 488 минус 78?";
$string2 = $string;
$string =~ s/\D*(\d+).*/$1/;
$string2 =~ s/\D*\d+\D*(\d+).*/$1/;
print "$string\n";
print "$string2\n";
print $string - $string2, "\n";

$ perl re8.pl
488
78
410
```

Следующие несколько программ демонстрируют те неприятности, которые могут возникнуть при использовании непроцитированных круглых скобок в регулярных выражениях, когда не было намерения заключать часть регулярного выражения в скобки. Первая из этих программ пытается найти соответствие круглым скобкам в строке с помощью непроцитированных круглых скобок в регулярном выражении и терпит неудачу. Регулярное выражение ag(e соответствует той же строке, которой соответствует регулярное выражение age, поскольку круглая скобка является специальным символом, поэтому все регулярное выражение не соответствует строке ag(ed).

```
$ perl -le 'if ("ag(ed)" =~ /ag(ed)/) {print "true";} else {print "false";}'
false
```

В регулярном выражении, приведенном в следующем примере, круглые скобки цитируются путем установки перед каждой из них обратного слэша, что заставляет Perl интерпретировать эти скобки как обычные символы. Поиск этого соответствия завершается успешно.

```
$ perl -le 'if ("ag(ed)" =~ /ag\(ed\)/) {print "true";} else {print "false";}'
true
```

Затем Perl обнаруживает в регулярном выражении непарную круглую скобку:

```
$ perl -le 'if ("ag(ed)" =~ /ag(e/) {print "true";} else {print "false";}'
Unmatched ( in regex; marked by <-- HERE in m/ag( <-- HERE e/ at -e line 1.
Непарная ( в регулярном выражении, помечена <-- HERE в m/ag( <-- HERE e/ в -e,
в строке 1
```

Когда круглая скобка будет процитирована, все встанет на свои места, и Perl найдет соответствие:

```
$ perl -le 'if ("ag(ed)" =~ /ag\(e/) {print "true";} else {print "false";}'
true
```

Модули CPAN

Хранилище CPAN (Comprehensive Perl Archive Network — Всеобъемлющий сетевой архив Perl) предоставляет на своем веб-сайте (www.cpan.org) документацию по языку Perl, ответы на наиболее часто задаваемые вопросы (FAQ), модули (см. раздел «Терминология» на с. 507) и сценарии. В этом хранилище содержится более чем 16 000 дистрибутивов (см. раздел «Терминология») и предоставляются ссылки, подписки и версии Perl, скомпилированные для работы под управлением различных операционных систем (переносы языка Perl). Одним из способов розыска модуля является посещение веб-страницы search.cpan.org и использование поля поиска или щелчок на названии одного из классов модулей, перечисленных на этой странице.

В этом разделе объясняется, как загрузить модуль из CPAN и как установить и запустить модуль. Perl предоставляет для модулей иерархическое пространство имен, разделяя компоненты имени двойными двоеточиями (::). В примере, приводимом в данном разделе, используется модуль по имени Timestamp::Simple, о котором можно прочитать и который можно загрузить с веб-страницы search.cpan.org/dist/Timestamp-Simple. Метка времени (timestamp) — это дата и время в формате YYYYMMDDHHMMSS

Чтобы воспользоваться каким-либо из модулей Perl, сначала нужно загрузить файл, в котором он находится. Для данного примера на веб-странице search.cpan.org/~shoop/Timestamp-Simple-1.01/Simple.pm в ее правой части имеется ссылка Download (Загрузка). Нужно щелкнуть на ссылке и сохранить файл в предполагаемом рабочем каталоге. Наличие прав привилегированного пользователя не потребуется вплоть до последней стадии этой процедуры — установки модуля.

Большинство модулей Perl распространяется в виде сжатых tar-файлов (см. раздел «Сжатые tar-файлы» на с. 83). Когда загруженный файл окажется в рабочем каталоге, его нужно распаковать:

```
$ tar xzvf Timestamp-Simple-1.01.tar.gz
Timestamp-Simple-1.01/
Timestamp-Simple-1.01/Simple.pm
Timestamp-Simple-1.01/Makefile.PL
Timestamp-Simple-1.01/README
Timestamp-Simple-1.01/test.pl
Timestamp-Simple-1.01/Changes
Timestamp-Simple-1.01/MANIFEST
Timestamp-Simple-1.01/ARTISTIC
Timestamp-Simple-1.01/GPL
Timestamp-Simple-1.01/META.yml
```

В файле README, который находится в только что созданном каталоге, обычно предоставляются инструкции по сборке и установке модуля. Этапы установки большинства модулей совпадают.

```
$ cd Timestamp-Simple-1.01
$ perl Makefile.PL
Checking if your kit is complete...          Проверка комплектности инструментария...
Looks good                                   Все в комплекте
Writing Makefile for Timestamp::Simple       Запись Makefile для Timestamp::Simple
```

Если собираемый модуль зависит от других модулей, не установленных на локальной системе, запуск команды perl Makefile.PL приведет к выводу одного или несколь-

ких предупреждений о несоблюдении предварительных условий. Этот шаг приводит к записи сборочного файла (makefile), даже если модулей не хватает. В таком случае следующий шаг будет неудачным, и перед тем как продолжить процесс, придется заняться сборкой и установкой отсутствующих модулей.

Следующий шаг заключается в запуске утилиты make, работающей с только что созданным сборочным файлом. После ввода команды `make` нужно запустить команду `make test`, чтобы убедиться в работоспособности модуля.

```
$ make
cp Simple.pm blib/lib/Timestamp/Simple.pm
Manifying blib/man3/Timestamp::Simple.3pm

$ make test
PERL_DL_NONLAZY=1 /usr/bin/perl "-Iblib/lib" "-Iblib/arch" test.pl
1..1
# Running under perl version 5.100000 for linux
# Current time local: Fri Sep  4 18:20:41 2009
# Current time GMT:   Sat Sep  5 01:20:41 2009
# Using Test.pm version 1.25
ok 1
ok 2
ok 3
```

И наконец, работая с root-правами, нужно установить модуль:

```
# make install
Installing /usr/local/share/perl/5.10.0/Timestamp/Simple.pm
Installing /usr/local/man/man3/Timestamp::Simple.3pm
Writing /usr/local/lib/perl/5.10.0/auto/Timestamp/Simple/.packlist
Appending installation info to /usr/local/lib/perl/5.10.0/perllocal.pod
```

После того как модуль будет установлен, для вывода документации по порядку его использования можно воспользоваться утилитой perldoc. Пример использования данной утилиты показан в разделе «Утилита perldoc» на с. 506.

В некоторых модулях имеются разделы SYNOPSIS. Если такой раздел имеется в установленном модуле, этот модуль можно протестировать, поместив код из раздела SYNOPSIS в файл и запустив его в качестве программы на Perl:

```
$ cat times.pl
use Timestamp::Simple qw(stamp);
print stamp, "\n";

$ perl times.pl
20090905182627
```

После этого модуль можно вставить в программу на Perl. В следующем примере модуль timestamp используется для создания уникального имени файла:

```
$ cat fn.pl
use Timestamp::Simple qw(stamp);

# Сохранение метки времени в переменной
$ts = stamp, "\n";

# Удаление года
$ts =~ s/..(.*)/\1/;
```

```
# Создание уникального имени файла
$fn = "myfile." . $ts;

# Открытие файла, запись в него и закрытие файла
open (OUTFILE, '>', "$fn");
print OUTFILE "Привет.\n";
close (OUTFILE);
```

```
$ perl fn.pl
$ ls myf*
myfile.0905183010
```

substr

Функцию **substr** можно использовать для удаления года вместо регулярного выражения. Для этого нужно заменить строку, которая начинается с `$ts =~`, следующей строкой. В новой строке функция **substr** получает значение строки, хранящейся в переменной `$ts`, начиная с позиции 4 и до конца строки:

```
$ts = substr ($ts, 4);
```

Примеры

В этом разделе приводится ряд типовых программ на Perl. Сначала нужно попытаться запустить эти программы в исходном виде, а затем внести в них изменения с целью более глубокого освоения программирования на Perl.

Первый пример запускается под управлением Linux и выводит список групп, которым принадлежит пользователь, чье имя задано в качестве аргумента. Без задания аргументов он выводит список групп, в которых состоит пользователь, запустивший программу. Хеш `%ENV` в программе на Perl содержит переменные среды окружения той оболочки, которая вызвала Perl. Ключами этого хеша являются имена переменных среды окружения, а значениями хеша — значения соответствующих переменных. В первой строке программы переменной `$user` присваивается имя пользователя. Функция **shift** (см. раздел «shift, push, pop и splice» на с. 518) извлекает значение первого аргумента командной строки и сдвигает все остальные аргументы, если таковые имеются. Если пользователь запускает программу с аргументом, этот аргумент присваивается переменной `$user`. Если в командной строке не будет ни одного аргумента, работа функции **shift** завершится отказом, и Perl выполнит инструкцию, которая следует за булевым оператором ИЛИ (`||`). Эта инструкция извлечет значение, связанное с ключом `USER` в хеше `%ENV`, которое является именем пользователя программы.

Прием вывода от процесса

Третья инструкция инициализирует массив `@list`. Хотя эта инструкция и не нужна, ее включение считается устоявшейся практикой, упрощающей чтение кода. Следующая инструкция открывает лексический описатель `$fh`. Замыкающий символ канала (`|`) в части *ссылка_на_файл* этой инструкции (см. раздел «Открытие файла и присваивание описателя» на с. 530) предписывает Perl передать предшествующую каналу командную строку оболочке на выполнение и принять стандартный вывод из команды, когда программа осуществляет чтение с использованием описателя файла. В данном случае команда использует утилиту **grep**, чтобы отфильтровать файл **/etc/group** и получить строки, содержащие имя пользователя, хранящие-

еся в переменной $user. Если Perl не может открыть описатель, инструкция die выводит сообщение об ошибке.

```
$ cat groupfind.pl
$user = shift || $ENV{"USER"};
say "Пользователь $user принадлежит следующим группам:";
@list = ();
open (my $fh, "grep $user /etc/group |") or die "Ошибка: $!\n";
while ($group = <$fh>) {
    chomp $group;
    $group =~ s/(.*?):.*/$1/;
    push @list, $group;
}
close $fh;
@slist = sort @list;
say "@slist";

$ perl groupfind.pl
Пользователь принадлежит следующим группам:
adm admin audio cdrom dialout dip floppy kvm lpadmin …
```

Структура while в программе groupfind.pl считывает строки из стандартного вывода команды grep и завершает свою работу, когда закончится выполнение команды grep. В каждой строке, полученной из файла /etc/group, сначала следует имя группы, затем двоеточие и вся остальная информация, включая имена пользователей, принадлежащих этой группе. Строка из этого файла имеет следующий вид:

```
sam:x:1000:max,zach,helen
```

Строка кода

```
$group =~ s/(.*?):.*/$1/;
```

использует регулярное выражение и подстановку для удаления из каждой строки всего, кроме имени группы. Регулярное выражение .*: привело бы к поиску жадного соответствия нулю или большему количеству символов, сопровождаемых двоеточием; установка за звездочкой вопросительного знака приводит к тому, что выражение задает поиск нежадного соответствия (см. раздел «Нежадные соответствия» на с. 541). Установка круглых скобок вокруг части выражения, соответствующего той строке, которую программе нужно вывести, позволяет Perl использовать строку, соответствующую регулярному выражению, в строке замены. Заключительные символы .* соответствуют оставшейся части строки. Perl заменяет $1 в строке замены той строкой, которая соответствует заключенной в скобки части регулярного выражения (той части, которая находится между круглыми скобками), и присваивает это значение (имя группы) переменной $group.

Инструкция chomp удаляет замыкающий разделитель строк (регулярное выражение не соответствует этому символу). Инструкция push добавляет значение переменной $group к концу массива @list. Если не использовать инструкцию chomp, то имя каждой группы появлялось бы в выводе отдельной строкой. После того как структура while завершит обработку ввода, получаемого от команды grep, инструкция sort выстроит по порядку элементы массива @list и присвоит результат переменной массива @slist. Заключительная инструкция выведет отсортированный список имен групп, которым принадлежит пользователь.

opendir и readdir

В следующем примере представлены функции opendir и readdir. Функция opendir открывает каталог примерно так же, как функция open открывает обычный файл. Она использует два аргумента: имя описателя каталога и имя открываемого каталога. Функция readdir читает имена файлов из открытого каталога.

В данном примере функция opendir открывает рабочий каталог (указанный с помощью символа .), используя лексический описатель каталога $dir. Если работа функции opendir завершится неудачно, Perl выполнит ту инструкцию, которая следует за оператором or: инструкция die отправляет сообщение об ошибке на стандартную ошибку и завершает выполнение программы. При открытом каталоге структура while осуществляет последовательный перебор имен файлов в каталоге, присваивая имена файлов, возвращаемые функцией readdir, лексической переменной $entry. Инструкция if выполняет инструкцию print только для тех файлов, которые являются каталогами (-d).

Функция print выводит имя каталога, если только его имя не является . или .. . Когда функция readdir прочитает все файлы в рабочем каталоге, она возвращает *false*, и управление передается инструкции, которая следует за блоком while. Функция closedir закрывает открытый каталог, а функция print выводит разделитель строк, за которым следует список каталогов, отображаемый программой.

```
$ cat dirs2a.pl
#!/usr/bin/perl
print "В рабочем каталоге содержатся следующие каталоги:\n";
opendir my $dir, '.' or die "Невозможно открыть каталог: $!\n";
while (my $entry = readdir $dir) {
    if (-d $entry) {
        print $entry, ' ' unless ($entry eq '.' || $entry eq '..');
    }
}
closedir $dir;
print "\n";

$ ./dirs2a.pl
В рабочем каталоге содержатся следующие каталоги:
two one
```

split

Функция split разбивает строку на подстроки, определенные с помощью разделителя. Для вызова функции split используется следующий синтаксис

*split (/**регулярное_выражение**/, **строка**);*

где в качестве разделителя используется *регулярное_выражение* (часто в этом качестве используется обычный символ), а *строка* — это разбиваемая строка. Как показано в следующем примере, возвращаемый функцией split список можно присвоить переменной массива.

Следующая программа запускается под управлением Linux и выводит имена пользователей с идентификационными номерами UID, перечисленными в файле /etc/passwd, которые больше или равны 100. Для чтения строк из файла passwd в переменную $user используется структура while, а для разбиения строки на подстроки, разделенные двоеточиями, используется функция split. В строке, начинающейся с @row, каждая из этих подстрок присваивается элементам массива @row. Выражение

инструкции if вычисляется в *true*, если в третьей подстроке (UID) содержится значение, большее или равное 100. Это выражение использует оператор числового сравнения >=, поскольку ведется сравнение двух чисел; при алфавитном сравнении будет использоваться оператор сравнения строк ge.

Инструкция print отправляет UID-номер и связанное с ним имя пользователя в файл, для ссылки на который используется описатель $sortout. Инструкция open устанавливает для этого описателя канал, который отправляет его вывод команде sort -n. Поскольку утилита sort (см. раздел «sort: Отображение содержимого файла в определенном порядке» на с. 74) ничего не выводит на экран, пока не будет закончен весь ввод данных, программа split3.pl не выводит ничего до тех пор, пока она не закроет описатель $sortout, что она делает по завершении чтения файла passwd.

```
$ cat split3.pl
#!/usr/bin/perl -w

open ($pass, "/etc/passwd");
open ($sortout, "| sort -n");
while ($user = <$pass>) {
    @row = split (/:/, $user);
    if ($row[2] >= 100) {
        print $sortout "$row[2] $row[0]\n";
        }
    }
close ($pass);
close ($sortout);

$ ./split3.pl
100 libuuid
101 syslog
102 klog
103 avahi-autoipd
104 pulse
...
```

В следующем примере массив @ARGV (см. раздел «@ARGV» на с. 532) используется для подсчета и вывода аргументов, с которыми был вызван этот пример. Структура foreach осуществляет последовательный перебор элементов массива @ARGV, в котором содержатся аргументы командной строки. Оператор преинкремента ++ перед выводом переменной $count увеличивает ее значение на единицу.

```
$ cat 10.pl
#!/usr/bin/perl -w

$count = 0;
$num = @ARGV;
print "В командную строку введено аргументов: $num, а именно:\n";
foreach $arg (@ARGV) {
    print ++$count, ". $arg\n";
    }

$ ./10.pl apple pear banana watermelon
```

```
В командную строку введено аргументов: 4, а именно:
1. apple
2. pear
3. banana
4. watermelon
```

Заключение

Первая версия Perl была написана Ларри Уоллом (Larry Wall) в 1987 году. С тех пор Perl существенно вырос как в объеме, так и в функциональности, превратившись в весьма популярный язык, используемый для обработки текста, администрирования систем, разработки программного обеспечения и многоцелевого программирования. Одним из наиболее ценных качеств Perl является его поддержка со стороны созданных независимыми разработчиками многих тысяч модулей, многие из которых хранятся в репозитории CPAN.

Утилита perldoc находит и выводит локальную документацию по Perl. Она также позволяет документировать программы, написанные на Perl, путем вывода строк в pod-формате (plain old documentation — простая документация), включаемых вами в программу.

Perl предоставляет три типа переменных: скаляры (сингулярные переменные, чьи имена начинаются с символа $), массивы (множественные переменные, чьи имена начинаются с символа @) и хеши (иначе называемые ассициативными массивами; множественные переменные, имена которых начинаются с символа %). Переменные массивов и хешей содержат списки, но массивы в отличие от хешей имеют упорядоченную структуру. Стандартные инструкции управления процессом выполнения программы позволяют изменять порядок выполнения инструкций в программе на Perl. Кроме этого, в программах на Perl могут использоваться подпрограммы, включающие переменные, локальные по отношению к этим подпрограммам (лексические переменные).

Регулярные выражения являются одной из сильных сторон Perl. Вдобавок ко всему, что доступно во многих утилитах, Perl предоставляет такие функции работы с регулярными выражениями, которые позволяют выполнять более сложную обработку строк.

Упражнения

1. Какие два разных способа можно использовать для включения в Perl предупреждений?
2. Чем отличается массив от хеша?
3. В каком из каждых примеров лучше использовать хеш, а в каком массив?
   1) Подсчет числа появлений IP-адреса в регистрационном файле.
   2) Генерирование списка пользователей, превысивших квоту дискового пространства, выделенную под отчет.
4. Напишите регулярное выражение, соответствующее следующей, заключенной в кавычки строке:

   ```
   He said, "Go get me the wrench," but I didn't hear him.
   ```

5. Напишите регулярное выражение, соответствующее IP-адресу в регистрационном файле.
6. Многие файлы конфигурации содержат массу комментариев, включая и закомментированные по умолчанию конфигурационные директивы. Напишите программу, удаляющую эти комментарии из конфигурационного файла.

Упражнения повышенной сложности

1. Напишите программу, которая удаляет файлы *~ и *.ico из иерархии каталога. (*Подсказка.* Воспользуйтесь модулем File::Find.)
2. Опишите программную ошибку, которая не будет фигурировать в предупреждениях, выводимых Perl.
3. Напишите программу на Perl, ведущую подсчет количества файлов с определенными расширениями имен, имеющихся в рабочем каталоге, и подсчитывающую количество байтов в этих файлах.
4. Опишите разницу между строками, заключенными в одинарные и двойные кавычки.
5. Напишите программу, копирующую все файлы с расширениями имен .ico, имеющиеся в иерархии каталога, в каталог по имени icons, находящийся в вашем личном каталоге. (*Подсказка.* Воспользуйтесь модулями File::Find и File::Copy.)
6. Напишите программу, проводящую анализ регистрационных файлов Apache. Выведите количество байт, полученное по запросу каждого пути. Проигнорируйте неудачные запросы страниц. Если имеется более десяти путей, выведите только первые десять.

   Ниже показана типовая строка из регистрационного файла доступа к Apache. Два числа, следующие за строкой HTTP/1.1, — это код ответа и счетчик байт. Код ответа 200 означает, что запрос был успешным. Если в поле счетчика байт стоит прочерк (-), он означает, что данные не передавались.

```
__DATA__
92.50.103.52 - - [19/Aug/2008:08:26:43 -0400] "GET /perl/automated-testing/next_active.gif
HTTP/1.1" 200 980 "http://example.com/perl/automated-testing/navigation_bar.htm"
"Mozilla/5.0 (X11; U; Linux x86_64; en-US; rv:1.8.1.6) Gecko/20061201 Firefox/3.0.0.6
(Ubuntu-feisty); Blazer/4.0"
```

Глава 12. Язык обработки шаблонов AWK

AWK является языком сканирования и обработки данных с помощью шаблонов, позволяющим вести поиск записей (в качестве которых обычно выступают строки), соответствующих указанным шаблонам, в одном или нескольких файлах. При обработке строк выполняются такие действия, как запись строк в стандартный вывод или приращение значения счетчика при каждом обнаружении соответствия шаблону. В отличие от *процедурных* языков, AWK является *языком, управляемым данными*: ему предоставляется описание данных для обработки и сообщается, что нужно делать с этими данными в случае их обнаружения.

AWK можно использовать для составления отчетов или для фильтрации текста. Он одинаково хорошо работает как с числами, так и с текстами. AWK обычно дает правильный результат и при работе со смешанными числовыми и текстовыми данными. Авторы AWK (а этот язык назван аббревиатурой, образованной от фамилий авторов — Alfred V. Aho, Peter J. Weinberger, Brian W. Kernighan) разработали свое детище с расчетом на простоту его использования. Для этого в исходной реализации они сознательно пожертвовали скоростью работы.

AWK перенял многие конструкции у языка программирования C и обладает следующими свойствами:

- Гибким форматом.
- Условным выполнением.
- Инструкциями цикла.
- Числовыми переменными.
- Строковыми переменными.
- Регулярными выражениями.
- Выражениями отношений.
- Такой же, как у C, функцией printf.
- Работой в режиме использования сопроцесса (только в gawk).
- Обменом данными по сети (только в gawk).

Синтаксис

В командной строке gawk используется следующий синтаксис:

gawk [ключи] [программа] [список_файлов]
gawk [ключи] –f файл_программы [список_файлов]

Утилита gawk получает ввод из файлов, указанных в командной строке, или из стандартного ввода. Дополнительная команда getline предоставляет более широкий выбор источников ввода и способов, с помощью которых gawk считывает вводимые данные (см.

далее раздел «getline: Управление вводом» на с. 559). При использовании сопроцесса (coprocess) gawk может взаимодействовать с другими программами или вести обмен данными по сети (см. раздел «Сопроцесс: Двунаправленный ввод-вывод» на с. 581; в awk или mawk этот режим работы недоступен). Вывод из gawk направляется на стандартный вывод.

Аргументы

В приведенном выше примере синтаксиса *программа* — это программа на gawk, включаемая в командную строку. *Файл_программы* — это имя файла, в котором содержится программа на gawk. Включение *программы* в командную строку дает возможность писать короткие gawk-программы, не создавая отдельного *файл_программы*. Чтобы оболочка не интерпретировала команды gawk как команды оболочки, программу нужно поместить в одинарные кавычки. Если поместить длинную или сложную программу в файл, это позволит сократить количество возможных ошибок и исключить повторный набор программы. *Список_файлов* содержит путевые имена обычных файлов, обрабатываемых с помощью gawk. Эти файлы являются файлами ввода. Если *список_файлов* не указан, gawk получает ввод из стандартного ввода или из источника, указанного в инструкции getline (см. раздел «getline: Управление вводом» на с. 559), или от сопроцесса (см. раздел «Сопроцесс: Двунаправленный ввод-вывод» на с. 581).

У ЯЗЫКА AWK ИМЕЕТСЯ НЕСКОЛЬКО РЕАЛИЗАЦИЙ

Первоначально язык AWK был реализован в UNIX в качестве утилиты awk. Большинство дистрибутивов Linux предоставляет утилиту gawk (GNU-реализацию awk) или mawk (более быстродействующую, усеченную версию awk). Mac OS X предоставляет утилиту awk. В данной главе дается описание утилиты gawk. Все приводимые в главе примеры, за исключением особо оговоренных, работают также в awk и mawk, исключения составляют примеры, использующие сопроцессы (см. далее раздел «Сопроцесс: Двунаправленный ввод-вывод» на с. 581). Утилита gawk легко устанавливается на большинстве дистрибутивов Linux. При работе с Mac OS X для решения этого вопроса следует зайти на веб-сайт gawk.darwinports.com. Полный список расширений gawk можно найти, пройдя по ссылке GNU EXTENSIONS на man-странице gawk или на info-странице, посвященной этой утилите.

Ключи

С ключами, перед которыми ставится двойной дефис (--), работает только утилита gawk. При работе с awk или mawk эти ключи недоступны.

`--field-separator` *fs*
`-F` *fs*

Использование *fs* в качестве значения разделителя полей (field separator) ввода (см. описание переменной `FS` в табл. 12.2).

`--file` *файл_программы*
`-f` *файл_программы*

Чтение gawk-программы из файла по имени *файл_программы*, а не из командной строки. Этот ключ можно указать в командной строке более одного раза. Примеры даны в разделе «ключ - -file».

`--help -W help`

Краткая справка по использованию gawk (только в gawk).

`--lint -W lint`

Предупреждение о потенциально ошибочных или непереносимых конструктивных элементах gawk (только в gawk).

`--posix -W posix`

Запуск POSIX-совместимой версии gawk. Этот ключ накладывает ряд ограничений, подробности которых даны в man-странице, посвященной gawk (только в gawk).

`--traditional -W traditional`

Игнорирование в gawk-программе новых функций GNU, позволяющее получить программу, совместимую с UNIX awk (только в gawk).

`--assign` *переменная=значение*
`-v` *переменная=значение*

Присваивание *значения переменной*. Присваивание осуществляется еще до выполнения gawk-программы и доступно внутри шаблона `BEGIN` (см. раздел «BEGIN и END» на с. 555). Этот ключ можно указать в командной строке более одного раза.

Примечания

Обратите внимание на предыдущий совет, касающийся реализаций языка AWK.

Для удобства пользования многие системы Linux предоставляют ссылку из каталога /bin/awk на каталог /bin/gawk или /bin/mawk. Благодаря этому программу можно запускать, используя любое из этих имен.

Основы языка

Программа на gawk (*программа*, набранная в командной строке или в *файле_программы*) состоит из одной и более строк, в которых содержится *шаблон* и (или) *действие* в следующем формате:

шаблон { действие }

Шаблон выбирает строки из ввода. Утилита gawk выполняет *действие* над всеми строками, выбранными *шаблоном*. Фигурные скобки, в которые заключено *действие*, позволяют gawk отличить его от *шаблона*. Если программная строка не содержит *шаблона*, gawk выбирает из ввода все строки. Если программная строка не содержит *действия*, gawk копирует выбранные строки на стандартный вывод.

Сначала gawk сравнивает первую строку ввода (из файла, принадлежащего *списку_файлов* или из стандартного ввода) с каждым *шаблоном* программы. Если *шаблон* выбирает строку (если определяется ее соответствие *шаблону*), gawk выполняет *действие*, связанное с этим *шаблоном*. Если строка не выбрана, gawk не выполняет *действие*. Когда gawk завершит сравнение первой строки ввода, он повторяет процесс для следующей строки ввода. Он продолжает этот процесс сравнения последовательных строк ввода, пока не прочитает весь ввод.

Если несколько *шаблонов* выбирают одну и ту же строку, gawk выполняет *действие*, связанное с каждым *шаблоном*, в порядке его появления в программе. Утилита gawk может отправить на стандартный вывод одну и ту же строку ввода более одного раза.

Шаблоны

~ и !~ В качестве *шаблона* можно использовать регулярное выражение, заключенное в слэши. Оператор ~ проверяет, соответствует ли поле или переменная регулярному выражению (см. примеры в разделе «~ (оператор соответствия)»). Оператор !~ проводит проверку на несоответствие. Используя операторы отношений, перечисленные в табл. 12.1, можно проводить как числовое, так и строковое сравнение. Булевые операторы || (ИЛИ) и && (И) позволяют объединять любые *шаблоны*.

Таблица 12.1. Операторы отношений

Оператор отношения	Значение
<	Меньше чем
<=	Меньше чем или равно
==	Равно
!=	Не равно
>=	Больше чем или равно
>	Больше чем

BEGIN и END Два уникальных шаблона, BEGIN и END, выполняют команды перед тем, как утилита gawk приступит к обработке ввода, и после того, как она завершит эту обработку. До начала обработки всего ввода утилита gawk выполняет *действия*, связанные с шаблоном BEGIN, а по окончании обработки — *действия*, связанные с шаблоном END. Соответствующие примеры даны далее в разделах «BEGIN» и «END» на с. 555.

, (запятая) Запятая является оператором диапазона. Если два *шаблона* на одной строке gawk-программы разделены запятой, gawk выбирает диапазон строк, начиная с первой строки, соответствующей первому *шаблону*. Последней строкой, выбранной gawk, будет следующая строка из последовательности строк, соответствующая второму *шаблону*. Если второму шаблону не будет соответствовать ни одна строка, gawk выберет каждую строку до конца ввода. После того как утилита gawk обнаружит соответствие второму *шаблону*, она возобновляет этот процесс, начиная новый поиск соответствия первому *шаблону*. Примеры даны в разделе «Примеры», в подразделе «, (оператор диапазона)» на с. 565.

Действия

Действие является частью команды gawk, заставляющее gawk выполнить это *действие* при обнаружении соответствия *шаблону*. Если *действие* не указано, gawk выполняет действие по умолчанию. В этом качестве используется команда print (которая в явном виде предоставляется как {print}). Это действие копирует запись (как правило, строку; см. далее раздел «Разделители записей» на с. 557) из ввода в стандартный вывод.

Когда за командой print указываются аргументы, gawk выводит только их значения. Аргументами могут быть переменные или строковые константы. Вывод команды print

можно отправить в файл (используя в gawk-программе символ >; см. далее раздел «Перенаправленый вывод» на с. 569), добавить его к файлу (>>) или отправить через канал на ввод другой программы (|). Сопроцесс (|&) является двунаправленным каналом, осуществляющим обмен данными с программой, запущенной в фоновом режиме (доступен только в gawk; см. дальше раздел «Сопроцесс: Двунаправленный ввод-вывод» на с. 581).

Пока объекты в команде print не будут разделены запятыми, gawk проводит их объединение. Запятые заставляют gawk разделить объекты, используя разделитель полей в выводе (OFS; как правило, в качестве разделителя используется пробел; см. табл. 12.2).

В одну строку могут быть включены несколько действий, разделенные точками с запятой.

Комментарии

Утилита gawk игнорирует ту часть программной строки, которая следует за знаком решетки (#). Предваряя комментарии этим символом, можно осуществлять документирование gawk-программы.

Переменные

Хотя объявлять переменные в gawk до их применения не требуется, при желании им можно присваивать начальные значения. Беззнаковые числовые переменные инициализируются значением 0; строковые переменные инициализируются пустой строкой. Кроме поддержки пользовательских переменных, gawk поддерживает программные переменные. Оба этих вида переменных (пользовательские и программные) можно использовать в тех частях gawk-программы, которые относятся к *шаблонам* и *действиям*. Некоторые программные переменные перечислены в табл. 12.2.

Таблица 12.2. Переменные

Переменная	Значение
$0	Текущая запись (в виде значения одной переменной)
$1-$n	Поля в текущей записи
FILENAME	Имя текущего файла ввода (нулевая строка для стандартного ввода)
FS	Разделитель полей во вводе (по умолчанию пробел или TAB; см. раздел «переменная FS» на с. 570)
NF	Количество полей в текущей записи
NR	Номер текущей записи (см. раздел «NR (номер записи)» на с. 567)
OFS	Разделитель полей в выводе (по умолчанию пробел; см. раздел «переменная OFS» на с. 568)
ORS	Разделитель записей в выводе (по умолчанию разделитель строк)
RS	Разделитель записей во вводе (по умолчанию разделитель строк)

Кроме инициализации переменных внутри программы для их инициализации в командной строке можно воспользоваться ключом --assign (-v). Эта возможность пригодится в том случае, если значение переменной от запуска к запуску gawk изменяется.

Разделители записей

По умолчанию записи во вводе и в выводе отделяются друг от друга разделителем строк. То есть gawk принимает каждую строку ввода за отдельную запись и добавляет к концу каждой записи в выводе разделитель строк. По умолчанию разделителями полей во вводе являются символы пробелов и табуляции, а в качестве разделителя полей в выводе используется пробел. Значение любого разделителя можно в любое время заменить новым значением, присвоив это значение переменной либо из программы, либо из командной строки с помощью ключа `--assign` (`-v`).

Функции

В табл. 12.3 перечислены некоторые функции, предоставляемые gawk для работы с числами и строками.

Таблица 12.3. Функции

Функция	Назначение
length(*str*)	Возвращение количества символов в строке *str*; без аргументов — возвращение количества символов в текущей записи (см. раздел «функция length» на с. 566)
int(*num*)	Возвращение целой части числа *num*
index(*str1*,*str2*)	Возвращение индекса строки *str2* в строке *str1*, или 0, если строка *str2* в строке *str1* отсутствует
split(*str*,*arr*,*del*)	Помещение элементов строки *str*, разделенных с помощью символа *del*, в элементы массива *arr* [1]...*arr* [*n*]; возвращение количества элементов в массиве
sprintf(*fmt*,*args*)	Форматирование *args* в соответствии со значением *fmt* и возвращение отформатированной строки; имитирование работы одноименной функции языка программирования C
substr(*str*,*pos*,*len*)	Возвращение из строки *str* подстроки, которая начинается с позиции *pos* и имеет длину *len* символов
tolower(*str*)	Возвращение копии строки *str*, в которой все буквы верхнего регистра заменены точно такими же буквами нижнего регистра
toupper(*str*)	Возвращение копии строки *str*, в которой все буквы нижнего регистра заменены точно такими же буквами верхнего регистра

Арифметические операторы

Арифметические операторы gawk, перечисленные в табл. 12.4, позаимствованы из языка C.

Таблица 12.4. Арифметические операторы

Оператор	Значение
**	Возведение значения выражения, предшествующего оператору, в степень, которая указывается значением выражения, следующего за оператором
*	Умножение значения выражения, предшествующего оператору, на значение выражения, следующего за оператором

продолжение ⇩

Таблица 12.4 (*продолжение*)

Оператор	Значение
/	Деление значения выражения, предшествующего оператору, на значение выражения, следующего за оператором
%	Получение остатка от деления значения выражения, предшествующего оператору, на значение выражения, следующего за оператором
+	Сложение значения выражения, предшествующего оператору, со значением выражения, следующего за оператором
-	Вычитание значения выражения, следующего за оператором, из значения выражения, предшествующего оператору
=	Присваивание значения выражения, следующего за оператором, переменной, имя которой предшествует оператору
++	Увеличение на единицу значения переменной, предшествующей оператору
--	Уменьшение на единицу значения переменной, предшествующей оператору
+=	Сложение значения выражения, следующего за оператором, со значением переменной, предшествующей оператору, и присваивание результата переменной, предшествующей оператору
-=	Вычитание значения выражения, следующего за оператором, из значения переменной, предшествующей оператору, и присваивание результата переменной, предшествующей оператору
*=	Умножение значения переменной, предшествующей оператору, на значение выражения, следующего за оператором, и присваивание результата переменной, предшествующей оператору
/=	Деление значения переменной, предшествующей оператору, на значение выражения, следующего за оператором, и присваивание результата переменной, предшествующей оператору
%=	Вычисление остатка от деления значения переменной, предшествующей оператору, на значение выражения, следующего за оператором. Присваивание результата переменной, предшествующей оператору

Ассоциативные массивы

Ассоциативный массив — одно из самых мощных средств gawk. В качестве индексов в этих массивах используются строки. Использование в качестве индексов числовых строк позволяет имитировать обычный массив. В Perl ассоциативный массив называется *хешем* (см. раздел «Переменные, представляющие собой хеши» на с. 519).

Для присваивания значения элементу ассоциативного массива используется следующий синтаксис:

массив[строка] = значение

где *массив* — это имя массива, *строка* — индекс элемента массива, которому присваивается значение, а *значение* — это значение, которое присваивается данному элементу.

Используя следующий синтаксис, при работе с ассоциативным массивом можно использовать управляющую структуру:

*for (**элемент** in массив) действие*

где *элемент* — это переменная, которая принимает значение каждого элемента массива в процессе последовательного перебора элементов, осуществляемого циклом, *массив* —

имя массива, а *действие* — это действие, которое gawk выполняет в отношении каждого элемента массива. В этом действии можно использовать переменную *элемент*.

Примеры программ, использующих ассоциативные массивы, даны в подразделе «ассоциативные массивы» раздела «Примеры» на с. 571 .

printf

Для управления форматом вывода, генерируемого gawk, вместо команды print можно использовать команду printf. Версия printf, используемая в gawk, похожа на одноименную функцию языка С. Команда printf использует следующий синтаксис:

printf "управляющая_строка",арг1, арг2, ..., аргn

Управляющая_строка определяет, как именно printf форматирует *арг1, арг2, ..., аргn*. Эти аргументы могут быть переменными или другими выражениями. Внутри *управляющей_строки* можно использовать символы \n для обозначения разделителя строк и \t для обозначения символа TAB. *Управляющая_строка* содержит спецификации преобразования, по одной для каждого аргумента. Для спецификации преобразования используется следующий синтаксис:

%[-][x[.y]]conv

где символ дефиса (-) заставляет printf выровнять аргумент по левому краю, *x* — это минимальная ширина поля, а *.y* — количество знакомест в числе справа от десятичной точки. Элемент *conv* обозначает тип числового преобразования и может быть выбран из букв, показанных в табл. 12.5. Примеры программ, использующих printf, показаны в подразделе «printf» раздела «Примеры»на с. 569.

Таблица 12.5. Преобразование чисел

conv	**Тип преобразования**
d	Десятичное число
e	Число с экспоненциальной формой записи
f	Число с плавающей точкой
g	Использование преобразования f или e, в зависимости от того, какое из них короче
o	Беззнаковое восьмеричное число
s	Строка символов
x	Беззнаковое шестнадцатеричное число

Управляющие структуры

Управляющие (изменяющие ход программы) инструкции изменяют порядок выполнения команд в gawk-программе. В этом разделе рассматриваются подробности управляющих структур if...else, while и for. Кроме этого рассматриваются инструкции break и continue, которые используются в управляющих структурах для изменения порядка выполнения команд. Дополнительные сведения об управляющих структурах даны в разделе «Управляющие структуры» на с. 417. При указании одной простой команды использовать вокруг нее фигурные скобки необязательно.

Управляющая структура if...else

Управляющая структура if...else проверяет состояние, возвращенное *условием*, и передает управление на основе этого состояния. В показанном ниже синтаксисе структуры if...else часть else является необязательной.

```
if (условие)
  {команды}
 [else
  {команды}]
```

В простой, показанной здесь инструкции if фигурные скобки не используются:

```
if ($5 <= 5000) print $0
```

Следующая gawk-программа использует простую структуру if...else. Здесь фигурные скобки также не используются.

```
$ cat if1
BEGIN {
    nam="sam"
    if (nam == "max")
            print "nam is max"
        else
            print "nam is not max, it is", nam
    }

$ gawk -f if1
nam is not max, it is sam
```

Управляющая структура while

Структура while осуществляет циклический проход и выполнение *команд* до тех пор, пока *условие* вычисляется в *true*. Для структуры while используется следующий синтаксис:

```
while (условие)
  {команды}
```

Следующая gawk-программа использует простую структуру while для вывода степеней числа два. В этом примере используются фигурные скобки, потому что цикл while состоит из более чем одной инструкции. Эта программа не получает ввода, и вся обработка проводится при выполнении gawk инструкций, связанных с шаблоном BEGIN.

```
$ cat while1
BEGIN{
    n = 1
    while (n <= 5)
        {
        print "2^" n, 2**n
        n++
        }
    }

$ gawk -f while1
2^1 2
2^2 4
```

```
2^3 8
2^4 16
2^5 32
```

Управляющая структура for

Для управляющей структуры for используется следующий синтаксис:

for (инициализация; условие; инкремент)
{команды}

Структура for начинает свою работу с выполнения инструкции *инициализации*, которая обычно устанавливает значение счетчика в 0 или 1. Затем она выполняет циклический проход *команд* до тех пор, пока *условие* вычисляется в *true*. После каждого цикла структура выполняет инструкцию *инкремент*. Gawk-программа for1 делает то же самое, что и предыдущая команда while1, с той лишь разницей, что в ней используется инструкция for, упрощающая код программы:

```
$ cat for1
BEGIN   {
        for (n=1; n <= 5; n++)
        print "2^" n, 2**n
        }

$ gawk -f for1
2^1 2
2^2 4
2^3 8
2^4 16
2^5 32
```

Утилита gawk поддерживает альтернативный синтаксис структуры for для работы с ассоциативными массивами:

for (переменная in массив)
{команды}

Эта структура for выполняет последовательный перебор элементов ассоциативного массива по имени *массив*, присваивая значение индекса каждого элемента *массива* переменной с именем *переменная* при каждом проходе цикла. В следующей строке кода (из программы подраздела «структура for» раздела «Примеры» на с. 571) демонстрируется структура for:

```
END {for (name in manuf) print name, manuf[name]}
```

Инструкция break

Инструкция break передает управление за пределы цикла for или цикла while, прерывая выполнение цикла, внутри которого она присутствует.

Инструкция continue

Инструкция continue передает управление в конец цикла for или цикла while, приводя к тому, что цикл, в котором она присутствует, продолжается со следующей итерации.

Примеры

Файл данных cars

Многие примеры этого раздела работают с файлом данных cars. Если смотреть слева направо, то столбцы в этом файле содержат название фирмы-производителя каждого автомобиля, модель, год изготовления, пробег в тысячах миль и цену. Все пустые пространства в этом файле сформированы исключительно из символов табуляции (а символы пробела в данном файле отсутствуют).

```
$ cat cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
ford    thundbd 2003    15      10500
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
ford    taurus  2004    10      17000
toyota  rav4    2002    180     750
chevy   impala  1985    85      1550
ford    explor  2003    25      9500
```

Отсутствие шаблона

Простая gawk-программа имеет следующий вид:

```
{ print }
```

Эта программа состоит из одной строки, представляющей собой *действие*. Поскольку *шаблон* отсутствует, gawk выбирает все строки ввода. При использовании без аргументов команда print выводит каждую выбранную строку целиком. Эта программа копирует ввод на стандартный вывод.

```
$ gawk '{ print }' cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
...
```

Отсутствие действия

В следующей программе есть *шаблон*, но нет явно указанного *действия*. Слэши показывают, что строка chevy является регулярным выражением.

```
/chevy/
```

В данном случае gawk выбирает из ввода только те строки, в которых содержится строковое значение chevy. Когда *действие* не указано, gawk предполагает, что в качестве *действия* нужно использовать команду print. В следующем примере на стандартный вывод копируются все строки ввода, в которых содержится строковое значение chevy:

```
$ gawk '/chevy/' cars
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
chevy   impala  1985    85      1550
```

Одинарные кавычки

Хотя использование одинарных кавычек в командной строке не требуется ни синтаксисом gawk, ни синтаксисом оболочки, их использование имеет вполне определенный смысл, поскольку они позволяют предотвратить возник-

новение ряда проблем. Если создаваемая в командной строке gawk-программа включает пробелы или символы, имеющие для оболочки специальное значение, эти символы нужно процитировать. Постоянное заключение программы в одинарные кавычки является наиболее простым способом обеспечения цитирования любых (нуждающихся в этом) символов.

Поля

В следующем примере из-за отсутствия *шаблона* из файла выбираются все строки. *Действие* заключено в фигурные скобки, оно всегда должно быть заключено в эти скобки, чтобы утилита gawk могла отличить его от *шаблона*. Этот пример выводит третье поле ($3), пробел (разделитель полей в выводе, обозначенный запятой) и первое поле ($1) каждой выбранной строки:

```
$ gawk '{print $3, $1}' cars
1970 plym
1999 chevy
1965 ford
1998 volvo
...
```

В следующем примере, который включает и *шаблон* и *действие*, выбираются все строки, содержащие строковое значение chevy, и из выбранных строк выводятся третьи и первые поля:

```
$ gawk '/chevy/ {print $3, $1}' cars
1999 chevy
2000 chevy
1985 chevy
```

В следующем примере gawk выбирает строки, которые соответствуют регулярному выражению h. Поскольку *действие* явным образом не указано, утилита gawk выводит все выбранные ею строки.

```
$ gawk '/h/' cars
chevy   malibu  1999    60      3000
ford    thundbd 2003    15      10500
chevy   malibu  2000    50      3500
honda   accord  2001    30      6000
chevy   impala  1985    85      1550
```

~ (оператор соответствия)

В следующем *шаблоне* для вывода всех строк, содержащих в первом поле букву h, используется оператор соответствия (~):

```
$ gawk '$1 ~ /h/' cars
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
honda   accord  2001    30      6000
chevy   impala  1985    85      1550
```

Знак вставки (^) в регулярном выражении заставляет его соответствовать началу строки или, в данном случае, началу первого поля:

```
$ gawk '$1 ~ /^h/' cars
honda   accord  2001    30      6000
```

Квадратные скобки являются ограничителями определения класса символов. В следующем примере утилита gawk выбирает строки, в которых имеется второе поле,

начинающееся с буквы t или с буквы m, и выводит третье и второе поля, знак доллара и пятое поле. Поскольку между "$" и $5 запятая не стоит, gawk в выводе между ними пробел не ставит.

```
$ gawk '$2 ~ /^[tm]/ {print $3, $2, "$"  $5}' cars
1999 malibu $3000
1965 mustang $10000
2003 thundbd $10500
2000 malibu $3500
2004 taurus $17000
```

знак доллара

В следующем примере показаны три разных роли, которые может играть знак доллара в gawk-программе. Во-первых, знак доллара, за которым следует число, обозначает поле. Во-вторых, внутри регулярного выражения знак доллара заставляет искать соответствие в конце строки или поля (5$). В-третьих, внутри строки знак доллара представляет себя самого.

```
$ gawk '$3 ~ /5$/ {print $3, $1, "$"  $5}' cars
1965 ford $10000
1985 bmw $450
1985 chevy $1550
```

В следующем примере оператор отношения «равно» (==) заставляет gawk проводить числовое сравнение третьего поля каждой строки и числа 1985. Команда gawk выполняет действие по умолчанию (print) для каждой строки, где сравнение вычисляется в *true*.

```
$ gawk '$3 == 1985' cars
bmw     325i    1985    115     450
chevy   impala  1985    85      1550
```

В следующем примере разыскиваются все автомобили стоимостью $3000 или меньшей.

```
$ gawk '$5 <= 3000' cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
bmw     325i    1985    115     450
toyota  rav4    2002    180     750
chevy   impala  1985    85      1550
```

текстовые сравнения

При использовании двойных кавычек gawk проводит текстовые сравнения, используя в качестве основы для такого сравнения таблицу ASCII-символов или другую локальную упорядоченную последовательность символов. В следующем примере gawk показывает, что строки 450 и 750 попадают в диапазон, который лежит между строками 2000 и 9000, что вряд ли является желаемым результатом.

```
$ gawk '"2000" <= $5 && $5 < "9000"' cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
toyota  rav4    2002    180     750
```

Когда необходимо провести числовое сравнение, кавычки использовать не нужно. В следующем примере выдается желаемый результат. Этот пример похож на предыдущий, за исключением того, что двойные кавычки в нем не используются.

```
$ gawk '2000 <= $5 && $5 < 9000' cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
honda   accord  2001    30      6000
```

, (оператор диапазона)

Оператор диапазона (,) выбирает группу строк. Первая выбираемая строка определяется *шаблоном*, стоящим перед запятой. Последняя выбираемая строка определяется *шаблоном*, стоящим после запятой. Если *шаблону*, стоящему после запятой, не будет соответствовать ни одна строка, gawk выбирает каждую строку до конца ввода. В следующем примере выбираются все строки, начинающиеся со строки, содержащей строковое значение volvo, и заканчивающиеся строкой, содержащей строковое значение bmw.

```
$ gawk '/volvo/ , /bmw/' cars
volvo   s80     1998    102     9850
ford    thundbd 2003    15      10500
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
```

После того как оператор диапазона найдет свою первую группу строк, он возобновляет процесс поиска строки, соответствующей *шаблону* перед запятой. В следующем примере gawk находит три группы строк, стоящих между chevy и ford.

Хотя в пятой строке ввода содержится слово ford, gawk не выбирает эту строку, поскольку ко времени обработки этой строки им ведется поиск слова chevy.

```
$ gawk '/chevy/ , /ford/' cars
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
ford    taurus  2004    10      17000
chevy   impala  1985    85      1550
ford    explor  2003    25      9500
```

ключ – –file

При написании длинной gawk-программы ее удобнее всего поместить в файл и сослаться на этот файл из командной строки. Для этого нужно воспользоваться ключом -f (--file), за которым следует имя файла, содержащего gawk-программу.

BEGIN

Следующая gawk-программа, сохраненная в файле под именем pr_header, содержит два *действия* и использует *шаблон* BEGIN. Утилита gawk выполняет действие, связанное с BEGIN, перед обработкой любых строк файла данных: она выводит заголовок.

Второе *действие*, {print}, не имеет *шаблона* и выводит все строки ввода.

```
$ cat pr_header
BEGIN   {print "Произв. Модель  Год     Пробег  Цена"}
        {print}
```

```
$ gawk -f pr_header cars
Произв. Модель  Год     Пробег  Цена
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
...
```

В следующем примере *действие*, связанное с *шаблоном* BEGIN, расширяется. В предыдущем и следующих примерах пустое пространство в заголовке составлено из отдельных символов табуляции, поэтому строка заголовков стоит точно над столбцами данных.

```
$ cat pr_header2
BEGIN   {
print " Произв. Модель  Год     Пробег  Цена"
print "---------------------------------------"
}
        {print}

$ gawk -f pr_header2 cars
Произв. Модель  Год     Пробег  Цена
---------------------------------------
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
...
```

функция length

При вызове функции length без аргументов она возвращает количество символов в текущей строке, включая разделители полей. В переменной $0 всегда содержится значение текущей строки. В следующем примере gawk ставит впереди каждой строки ее длину, а затем канал отправляет вывод из gawk команде sort (ключ -n указывает на числовую сортировку; см. описание команды sort в части V). В результате строки файла cars появляются в порядке их длины.

```
$ gawk '{print length, $0}' cars | sort -n
21 bmw  325i    1985    115     450
22 plym fury    1970    73      2500
23 volvo        s80     1998    102     9850
24 ford explor  2003    25      9500
24 toyota       rav4    2002    180     750
25 chevy        impala  1985    85      1550
25 chevy        malibu  1999    60      3000
25 chevy        malibu  2000    50      3500
25 ford taurus  2004    10      17000
25 honda        accord  2001    30      6000
26 ford mustang 1965    45      10000
26 ford thundbd 2003    15      10500
```

На формат этого отчета влияет использование символов табуляции для горизонтального выравнивания. Три дополнительных символа в начале каждой строки нарушают формат нескольких строк; средство исправления этого дефекта будет рассмотрено ниже.

NR (номер записи) Переменная NR содержит номер записи (строки), принадлежащий текущей строке. Следующий *шаблон* выбирает все строки, содержащие более 24 символов. *Действие* выводит номер каждой выбранной строки.

```
$ gawk 'length > 24 {print NR}' cars
2
3
5
6
8
9
11
```

Чтобы вывести группу строк файла на основе их номеров строк, оператор диапазона (,) и переменную NR можно объединить. В следующем примере выводятся строки со второй по четвертую:

```
$ gawk 'NR == 2 , NR == 4' cars
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
```

END *Шаблон* END работает так же, как и шаблон BEGIN, за исключением того, что gawk выполняет *действия*, связанные с этим шаблоном, после обработки последней строки ввода. Следующий отчет выводиться только после того, как будет обработан весь ввод. Переменная NR сохраняет свое значение после того как gawk закончит обработку файла данных, поэтому она может использоваться в *действии*, связанном с *шаблоном* END.

```
$ gawk 'END {print NR, "автомобилей на продажу." }' cars
12 автомобилей на продажу.
```

В следующем примере, чтобы раскрыть аббревиатуру, используемую в некоторых первых полях, используется управляющая структура if. Пока утилита gawk не изменяет запись, она оставляет всю ее (включая разделители) в неизменном виде. При изменении записи gawk заменяет все разделители в этой записи на значение разделителя полей в выводе. По умолчанию таким разделителем является пробел.

```
$ cat separ_demo
        {
        if ($1 ~ /ply/)  $1 = "plymouth"
        if ($1 ~ /chev/) $1 = "chevrolet"
        print
        }

$ gawk -f separ_demo cars
plymouth fury 1970 73 2500
chevrolet malibu 1999 60 3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
ford    thundbd 2003    15      10500
chevrolet malibu 2000 50 3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
ford    taurus  2004    10      17000
```

```
toyota  rav4    2002    180     750
chevrolet impala 1985 85 1550
ford    explor  2003    25      9500
```

автономный сценарий

Вместо вызова gawk из командной строки с ключом `-f` и именем запускаемой программы, можно написать сценарий, вызывающий gawk с командами, которые нужно запустить. В следующем примере показан автономный сценарий, запускающий ту же программу, что и в предыдущем примере. Команда `#!/bin/gawk -f` (см. раздел «#! — способ указания оболочки» на с. 294) осуществляет непосредственный запуск утилиты gawk. Чтобы выполнить эту команду, у вас должны быть права как на чтение, так и на выполнение файла сценария (см. раздел «chmod: Превращение файла в исполняемый» на с. 292).

```
$ chmod u+rx separ_demo2
$ cat separ_demo2
#!/bin/gawk -f
        {
        if ($1 ~ /ply/)  $1 = "plymouth"
        if ($1 ~ /chev/) $1 = "chevrolet"
        print
        }

$ ./separ_demo2 cars
plymouth fury 1970 73 2500
chevrolet malibu 1999 60 3000
ford    mustang 1965    45      10000
...
```

переменная OFS

Значение разделителя полей в выводе можно изменить, присвоив его значение переменной `OFS`. В следующем примере переменной `OFS` присваивается значение символа табуляции, для чего используется escape-последовательность с символом обратного слэша `\t`. Эта поправка корректирует внешний вид отчета, но столбцы по вертикали должным образом не выравнивает.

```
$ cat ofs_demo
BEGIN {OFS = "\t"}
    {
    if ($1 ~ /ply/)  $1 = "plymouth"
    if ($1 ~ /chev/) $1 = "chevrolet"
    print
    }

$ gawk -f ofs_demo cars
plymouth        fury    1970    73      2500
chevrolet       malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
ford    thundbd 2003    15      10500
chevrolet       malibu  2000    50      3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
ford    taurus  2004    10      17000
```

```
toyota  rav4    2002    180     750
chevrolet       impala  1985    85      1550
ford    explor  2003    25      9500
```

printf

Функцию printf можно использовать для уточнения формата вывода. В следующем примере для цитирования следующего разделителя строк в конце двух строк программы используется символ обратного слэша. Эту технологию можно использовать для продолжения длинной строки на одной или нескольких строках, что не влияет на результат работы программы.

```
$ cat printf_demo
BEGIN {
    print "                                    Мили"
    print "Произв.    Модель      Год     (X 1000) Цена"
    print \
    "-----------------------------------------------"
    }
    {
    if ($1 ~ /ply/)  $1 = "plymouth"
    if ($1 ~ /chev/) $1 = "chevrolet"
    printf "%-10s %-8s    %2d   %5d     $ %8.2f\n",\
        $1, $2, $3, $4, $5
    }

$ gawk -f printf_demo cars
                                   Мили
Произв.    Модель      Год     (X 1000) Цена
-------------------------------------------------
plymouth   fury        1970      73     $  2500.00
chevrolet  malibu      1999      60     $  3000.00
ford       mustang     1965      45     $ 10000.00
volvo      s80         1998     102     $  9850.00
ford       thundbd     2003      15     $ 10500.00
chevrolet  malibu      2000      50     $  3500.00
bmw        325i        1985     115     $   450.00
honda      accord      2001      30     $  6000.00
ford       taurus      2004      10     $ 17000.00
toyota     rav4        2002     180     $   750.00
chevrolet  impala      1985      85     $  1550.00
ford       explor      2003      25     $  9500.00
```

Перенаправленный вывод

В следующем примере создаются два файла: один со строками, содержащими chevy, а другой со строками, содержащими ford.

```
$ cat redirect_out
/chevy/ {print > "chevfile"}
/ford/ {print > "fordfile"}
END {print "Готово."}

$ gawk -f redirect_out cars
Готово.

$ cat chevfile
```

```
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
chevy   impala  1985    85      1550
```

Программа summary создает сводный отчет по всем, а также отчет по самым новым автомобилям. Несмотря на то что это формально не требуется, инициализация переменных в начале программы является правилом хорошего тона; на самом деле gawk автоматически объявляет и инициализирует переменные по мере их использования. После чтения всех данных ввода gawk вычисляет и выводит среднестатистические данные.

```
$ cat summary
BEGIN {
    yearsum = 0 ; costsum = 0
    newcostsum = 0 ; newcount = 0
    }
    {
    yearsum += $3
    costsum += $5
    }
$3 > 2000 {newcostsum += $5 ; newcount ++}
END {
    printf "Средний возраст машин (в годах) %4.1f \n",\
        2010 - (yearsum/NR)
    printf "Средняя цена машин $%7.2f\n",\
        costsum/NR
        printf "Средняя цена самых новых машин $%7.2f\n",\
            newcostsum/newcount
    }
```

```
$ gawk -f summary cars
Средний возраст машин (в годах) 17.1
Средняя цена машин $6216.67
Средняя цена самых новых машин $8750.00
```

Следующая команда gawk показывает формат строки из файла passwd операционной системы Linux, который используется в следующем примере:

```
$ awk '/mark/ {print}' /etc/passwd
mark:x:107:100:ext 112:/home/mark:/bin/tcsh
```

переменная FS

В следующем примере демонстрируется технология поиска в поле наибольшего числа. Поскольку работа ведется с файлом passwd операционной системы Linux, использующем в качестве разделителей полей двоеточие (:), перед чтением данных в этом примере осуществляется замена разделителя полей ввода (FS). Программа читает файл passwd и определяет следующий доступный пользовательский ID-номер (поле 3). Чтобы эта программа работала, номера в файле passwd не обязательно должны следовать по порядку.

Шаблон (`$3 > saveit`) заставляет gawk выбирать записи, которые содержат пользовательский ID-номер больше любого предыдущего обработанного им пользовательского ID-номера. При каждом выборе записи gawk присваивает значение нового пользовательского ID-номера переменной `saveit`. Затем gawk использует новое значение `saveit` для проверки пользовательских ID-номеров всех следующих записей. И наконец, gawk добавляет единицу к значению `saveit` и выводит результат.

```
$ cat find_uid
BEGIN             {FS = ":"
                  saveit = 0}
$3 > saveit          {saveit = $3}
END                  {print "Следующий доступный UID - " saveit + 1}

$ gawk -f find_uid /etc/passwd
Следующий доступный UID - 1092
```

Следующий пример создает еще один отчет на основе данных файла cars. Этот отчет использует вложенную структуру управления **if...else** для подстановки значения на основе содержимого поля price. *Шаблон* в программе не используется, поэтому она обрабатывает каждую запись.

```
$ cat price_range
    {
    if              ($5 <= 5000)                $5 = "недорогая"
        else if     (5000 < $5 && $5 < 10000)   $5 = "цена договорная"
        else if     (10000 <= $5)               $5 = "дорогая"
    #
    printf "%-10s %-8s    %2d    %5d    %-12s\n",\
    $1, $2, $3, $4, $5
    }

$ gawk -f price_range cars
plym       fury        1970       73    недорогая
chevy      malibu      1999       60    недорогая
ford       mustang     1965       45    дорогая
volvo      s80         1998      102    цена договорная
ford       thundbd     2003       15    дорогая
chevy      malibu      2000       50    недорогая
bmw        325i        1985      115    недорогая
honda      accord      2001       30    цена договорная
ford       taurus      2004       10    дорогая
toyota     rav4        2002      180    недорогая
chevy      impala      1985       85    недорогая
ford       explor      2003       25    цена договорная
```

ассоциативные массивы

Следующий ассоциативный массив manuf использует в качестве индекса содержимое первого поля каждой записи из файла cars. Массив состоит из элементов manuf[plym], manuf[chevy], manuf[ford] и т. д. Каждый новый элемент по мере создания инициализируется нулевым значением (0). Оператор ++ увеличивает значение переменной, за которой следует, на единицу.

структура for

Действие, следующее за *шаблоном* END, является структурой **for**, которая осуществляет последовательный перебор элементов ассоциативного массива. Канал пропускает вывод через команду **sort** для создания алфавитного списка автомобилей и указания их количества, имеющегося в наличии. Поскольку это сценарий оболочки, а не программный файл gawk, у вас должно быть право как на чтение, так и на выполнение файла manuf.

```
$ cat manuf
gawk ' {manuf[$1]++}
```

```
END {for (name in manuf) print name, manuf[name]}
' cars |
sort

$ ./manuf
bmw 1
chevy 3
ford 4
honda 1
plym 1
toyota 1
volvo 1
```

Следующая программа, **manuf.sh**, является сценарием оболочки более широкого назначения и содержит проверку на отсутствие ошибки. Это сценарий выводит список и ведет подсчет содержимого столбца в файле, при этом в командной строке указываются номер столбца и имя файла.

Первое *действие* (которое начинается со строки {count) использует в середине gawk-программы переменную оболочки $1 для указания индекса массива. Благодаря способу расстановки пар одинарных кавычек переменная $1, которая, казалось бы, появляется внутри одинарных кавычек, оказывается вне этих кавычек: ее окружают, но не включают в себя две заключенные в кавычки строки. Поскольку переменная $1 не взята в кавычки и находится в сценарии оболочки, оболочка выполняет подстановку вместо $1 значения первого аргумента командной строки (см. раздел «$1–$n: Аргументы командной строки» на с. 460). В результате этого переменная $1 интерпретируется еще до вызова команды gawk. Первый знак доллара (тот, что стоит в строке перед первой одинарной кавычкой) заставляет gawk интерпретировать значение, подставляемое оболочкой, в качестве номера поля.

```
$ cat manuf.sh
if [ $# != 2 ]
    then
        echo "Порядок использования: manuf.sh поле файл"
        exit 1
fi
gawk < $2 '
        {count[$'$1']++}
END         {for (item in count) printf "%-20s%-20s\n",\
            item, count[item]}' |
sort

$ ./manuf.sh
Порядок использования: manuf.sh поле файл

$ ./manuf.sh 1 cars
bmw                 1
chevy               3
ford                4
honda               1
plym                1
toyota              1
```

```
volvo                1

$ ./manuf.sh 3 cars
1965                 1
1970                 1
1985                 2
1998                 1
1999                 1
2000                 1
2001                 1
2002                 1
2003                 2
2004                 1
```

Способ, позволяющий избавиться от трюка с использованием кавычек и получить расширение параметров внутри gawk-программы, заключается в использовании в командной строке ключа –v для передачи gawk номера в виде переменной. Такая модификация позволит упростить чтение и отладку сценария для другого программиста. Сценарий manuf2.sh вызывается тем же способом, которым вызывался сценарий manuf.sh:

```
$ cat manuf2.sh
if [ $# != 2 ]
        then
                echo "Порядок использования: manuf2.sh поле файл"
                exit 1
fi
gawk -v "field=$1" < $2 '
                {count[$field]++}
END             {for (item in count) printf "%-20s%-20s\n",\
                        item, count[item]}' |
sort
```

Сценарий word_usage выводит частотный список использования слов в файле, указанном в командной строке. Утилита tr (см. соответствующий раздел в части V) построчно выводит слова из стандартного ввода. Утилита sort перестраивает файл, помещая вначале наиболее часто используемые слова. Сценарий выстраивает группы слов, используемые одинаковое количество раз, в алфавитном порядке.

```
$ cat word_usage
tr -cs 'a-zA-Z' '[\n*]' < $1 |
gawk
        {count[$1]++}
END         {for (item in count) printf "%-15s%3s\n", item, count[item]}' |
sort +1nr +0f -1

$ ./word_usage textfile
the             42
file            29
fsck            27
system          22
you             22
to              21
```

```
it              17
SIZE            14
and             13
MODE            13
...
```

Следующая программа аналогична предыдущей, но использует другой формат кода. Стиль имитирует программу на языке C, что упрощает чтение и разработку более сложных программ на gawk.

```
$ cat word_count
tr -cs 'a-zA-Z' '[\n*]' < $1 |
gawk '    {
        count[$1]++
}
END         {
        for (item in count)
            {
            if (count[item] > 4)
                {
                printf "%-15s%3s\n", item, count[item]
                }
        }
} ' |
sort +1nr +0f -1
```

Утилита tail выводит последние десять строк вывода, показывая, что слова, появляющиеся более пяти раз, в список не попадают:

```
$ ./word_count textfile | tail
directories      5
if               5
information      5
INODE            5
more             5
no               5
on               5
response         5
this             5
will             5
```

В следующем примере показан один из способов помещения данных в отчет. Первая строка ввода в gawk-программу берется из команды date. Программа читает эту строку как запись номер 1 (NR == 1), обрабатывает ее соответствующим образом, а все остальные строки обрабатывает с помощью *действия*, связанного со следующим шаблоном (NR > 1).

```
$ cat report
if (test $# = 0) then
    echo "Необходимо указать имя файла."
    exit 1
fi
(date; cat $1) |
gawk '
NR == 1 {print "Отчет по состоянию на ", $1, $2, $3 ", " $6}
```

```
NR >  1 {print $5 "\t" $1}'

$ ./report cars
Отчет по состоянию на Mon Jan 31, 2010
2500    plym
3000    chevy
10000   ford
9850    volvo
10500   ford
3500    chevy
450     bmw
6000    honda
17000   ford
750     toyota
1550    chevy
9500    ford
```

В следующем примере суммируется каждый столбец файла, указанного в командной строке. Программа получает ввод из файла numbers и осуществляет проверку на отсутствие ошибки, сообщает о строках, не содержащих числовые записи, и не включает их в отчет. В ней используется команда next (с комментарием «пропуск неподходящих записей»), чтобы все остальные команды для текущей записи, не содержащей числовой элемент, были пропущены. В конце программы gawk выводит для файла общую сумму чисел.

```
$ cat numbers
10      20      30.3    40.5
20      30      45.7    66.1
30      xyz     50      70
40      75      107.2   55.6
50      20      30.3    40.5
60      30      45.0    66.1
70      1134.7  50      70
80      75      107.2   55.6
90      176     30.3    40.5
100     1027.45 45.7    66.1
110     123     50      57a.5
120     75      107.2   55.6

$ cat tally
gawk ' BEGIN          {
                ORS = ""
                }
NR == 1 {                       # только для первой записи
    nfields = NF                # присваивание nfields количества
    }                           # полей в записи (NF)

    if ($0 ~ /[^0-9. \t]/)      # проверка каждой записи на содержание
        {                       # любых нечисловых символов,
        print "\nЗапись " NR " пропущена:\n\t"
                                # точек, пробелов или символов TAB
        print $0 "\n"
```

```
        next                    # пропуск неподходящих записей
        }
    else
        {
        for (count = 1; count <= nfields; count++)
                                # перебор всех полей подходящих записей
            {
            printf "%10.2f", $count > "tally.out"
            sum[count] += $count
            gtotal += $count
            }
        print "\n" > "tally.out"
        }
    }
END         {                   # после обработки последней записи
    for (count = 1; count <= nfields; count++)
                                # вывод суммарного значения
        {
        print "   -------" > "tally.out"
        }
    print "\n" > "tally.out"
    for (count = 1; count <= nfields; count++)
        {
        printf "%10.2f", sum[count] > "tally.out"
        }
    print "\n\n        Общая сумма " gtotal "\n" > "tally.out"
} ' < numbers

$ ./tally
Запись 3 пропущена:
        30      xyz     50      70
Запись 6 пропущена:
        60      30      45.0    66.1
Запись 11 пропущена:
        110     123     50      57a.5

$ cat tally.out
     10.00     20.00     30.30     40.50
     20.00     30.00     45.70     66.10
     40.00     75.00    107.20     55.60
     50.00     20.00     30.30     40.50
     70.00   1134.70     50.00     70.00
     80.00     75.00    107.20     55.60
     90.00    176.00     30.30     40.50
    100.00   1027.45     45.70     66.10
    120.00     75.00    107.20     55.60
   -------   -------   -------   -------
    580.00   2633.15    553.90    490.50
        Общая сумма 4257.55
```

В следующем примере программа читает файл passwd, выводит список пользователей, не имеющих пароль, и пользователей с продублированными ID-номерами.

(Точно такую же проверку проводит имеющаяся в Linux утилита pwck.) Поскольку в Mac OS X вместо файла passwd используется Open Directory, под управлением OS X этот пример работать не будет.

```
$ cat /etc/passwd
bill::102:100:ext 123:/home/bill:/bin/bash
roy:x:104:100:ext 475:/home/roy:/bin/bash
tom:x:105:100:ext 476:/home/tom:/bin/bash
lynn:x:166:100:ext 500:/home/lynn:/bin/bash
mark:x:107:100:ext 112:/home/mark:/bin/bash
sales:x:108:100:ext 102:/m/market:/bin/bash
anne:x:109:100:ext 355:/home/anne:/bin/bash
toni::164:100:ext 357:/home/toni:/bin/bash
ginny:x:115:100:ext 109:/home/ginny:/bin/bash
chuck:x:116:100:ext 146:/home/chuck:/bin/bash
neil:x:164:100:ext 159:/home/neil:/bin/bash
rmi:x:118:100:ext 178:/home/rmi:/bin/bash
vern:x:119:100:ext 201:/home/vern:/bin/bash
bob:x:120:100:ext 227:/home/bob:/bin/bash
janet:x:122:100:ext 229:/home/janet:/bin/bash
maggie:x:124:100:ext 244:/home/maggie:/bin/bash
dan::126:100::/home/dan:/bin/bash
dave:x:108:100:ext 427:/home/dave:/bin/bash
mary:x:129:100:ext 303:/home/mary:/bin/bash

$ cat passwd_check
gawk < /etc/passwd '     BEGIN   {
    uid[void] = ""              # сообщение для gawk, что uid - массив
    }
    {                           # шаблона нет; обрабока всех записей
    dup = 0                     # инициализация флажка дублирования
    split($0, field, ":")       # разбиение на поля, разделенные ":"
    if (field[2] == "")         # проверка на нулевую строку поля пароля
        {
        if (field[5] == "")     # проверка на нулевую строку поля info
            {
            print field[1] " не имеет пароля."
            }
        else
            {
            print field[1] " ("field[5]") не имеет пароля."
            }
        }
    for (name in uid)           # перебор элементов массива uid
        {
        if (uid[name] == field[3])
                                # проверка на повторное использование UID
            {
            print field[1] " имеет тот же UID, что и " name " : UID = " uid[name]
            dup = 1             # установка флажка дублирования
            }
        }
```

```
if (!dup)                       # то же самое, что и if (dup == 0)
                                # присваивание UID и имени входа в
                                # систему элементу массива uid
    {
    uid[field[1]] = field[3]
    }
}'
```

```
$ ./passwd_check
bill (ext 123) не имеет пароля.
toni (ext 357) не имеет пароля.
neil имеет тот же UID, что и toni : UID = 164
dan не имеет пароля.
dave имеет тот же UID, что и sales : UID = 108
```

В следующем примере показан полностью интерактивный сценарий оболочки, использующий gawk для генерации отчета по файлу cars на основе ценовых диапазонов:

```
$ cat list_cars
trap 'rm -f $$.tem > /dev/null;echo $0 aborted.;exit 1' 1 2 15
echo -n "Ценовой диапазон (например, 5000 7500):"
read lowrange hirange

echo '
                                Мили
Произв.    Модель      Год      (x 1000)   Цена
-----------------------------------------------' > $$.tem
gawk < cars '
$5 >= '$lowrange' && $5 <= '$hirange' {
        if ($1 ~ /ply/)  $1 = "plymouth"
        if ($1 ~ /chev/) $1 = "chevrolet"
        printf "%-10s %-8s    %2d    %5d    $ %8.2f\n", $1, $2, $3, $4,
$5
        }' | sort -n +5 >> $$.tem
cat $$.tem
rm $$.tem
```

```
$ ./list_cars
Ценовой диапазон (например, 5000 7500):3000 8000
                                Мили
Произв.    Модель      Год      (x 1000)   Цена
-----------------------------------------------
chevrolet  malibu      1999       60    $  3000.00
chevrolet  malibu      2000       50    $  3500.00
honda      accord      2001       30    $  6000.00

$ ./list_cars
Ценовой диапазон (например, 5000 7500):0 2000
                                Мили
Произв.    Модель      Год      (x 1000)   Цена
-----------------------------------------------
bmw        325i        1985      115    $   450.00
toyota     rav4        2002      180    $   750.00
```

```
chevrolet  impala      1985       85     $  1550.00

$ ./list_cars
Ценовой диапазон (например, 5000 7500):15000 100000
                                   Мили
Произв.    Модель      Год      (x 1000)   Цена
-----------------------------------------------
ford       taurus      2004       10     $ 17000.00
```

Расширенное программирование на gawk

В этом разделе рассматриваются некоторые расширенные возможности AWK. К ним можно отнести способы управления вводом с помощью инструкции **getline**, способы использования сопроцессов для обмена информацией между **gawk** и программой, работающей в фоновом режиме, а также способы использования сопроцесса для обмена данными по сети. Сопроцессы поддерживаются только в **gawk**, а в **awk** и **mawk** они не доступны.

getline: Управление вводом

Использование инструкции **getline** расширяет возможности управления данными, которые считываются **gawk**, по сравнению с теми возможностями, которые предоставляются другими методами ввода. Когда в качестве аргумента **getline** предоставляется имя переменной, эта инструкция считывает данные в эту переменную. Блок *шаблона* BEGIN в программе **g1** использует **getline** для чтения одной строки со стандартного ввода в переменную aa:

```
$ cat g1
BEGIN   {
        getline aa
        print aa
        }
$ echo aaaa | gawk -f g1
aaaa
```

В нескольких следующих примерах используется файл **alpha**:

```
$ cat alpha
aaaaaaaaa
bbbbbbbbb
ccccccccc
ddddddddd
```

Даже когда программе **g1** предоставляется более одной строки ввода, она все равно считывает только одну первую строку:

```
$ gawk -f g1 < alpha
aaaaaaaaa
```

Когда инструкции **getline** аргумент не предоставляется, она считывает ввод в переменную $0 и модифицирует переменные полей ($1, $2, ...):

```
$ gawk 'BEGIN {getline;print $1}' < alpha
aaaaaaaaa
```

В программе **g2** в блоке BEGIN для последовательного перебора строк в стандартном вводе использует цикл **while**. Инструкция **getline** считывает каждую строку в переменную holdme и выводит на экран каждое значение этой переменной.

```
$ cat g2
BEGIN    {
           while (getline holdme)
               print holdme
         }
```

```
$ gawk -f g2 < alpha
aaaaaaaaa
bbbbbbbbb
ccccccccc
ddddddddd
```

Программа g3 демонстрирует возможность gawk автоматически считывать каждую строку ввода в переменную $0, когда инструкция содержится в теле gawk-программы (а не только в блоке BEGIN). Эта программа выводит номер записи (NR), строку $0: и значение переменной $0 (текущую запись) для каждой строки ввода.

```
$ cat g3
        {print NR, "$0:", $0}

$ gawk -f g3 < alpha
1 $0: aaaaaaaaa
2 $0: bbbbbbbbb
3 $0: ccccccccc
4 $0: ddddddddd
```

Следующая программа, g4, показывает, что инструкция getline работает независимо от предоставляемого gawk автоматического чтения и независимо от переменной $0. Когда инструкция getline читает данные в переменную, она не изменяет ни переменную $0, ни какие-либо поля в текущей записи ($1, $2, ...). Первая инструкция в программе g4, которая совпадает с инструкцией в g3, выводит строку, автоматически прочитанную gawk.

Инструкция getline считывает следующую строку ввода в переменную по имени aa. Третья инструкция выводит номер записи, строку aa: и значение переменной aa. Вывод программы g4 показывает, что инструкция getline обрабатывает записи независимо от автоматического считывания, осуществляемого gawk.

```
$ cat g4
        {
        print NR, "$0:", $0
        getline aa
        print NR, "aa:", aa
        }

$ gawk -f g4 < alpha
1 $0: aaaaaaaaa
2 aa: bbbbbbbbb
3 $0: ccccccccc
4 aa: ddddddddd
```

Программа g5 выводит каждую строку ввода, за исключением тех строк, которые начинаются с буквы b. Первая инструкция print выводит каждую строку, прочитанную gawk в автоматическом режиме. Затем *шаблон* /^b/ выбирает все строки, начинающиеся с b, для специальной обработки. *Действие* использует getline для чтения следующей строки ввода в переменную hold, выводит строку «пропуск этой строки:», за которой следует значение переменной hold, и выводит значение переменной $1. В переменной $1 содержится значение первого поля той записи, которую утилита gawk считала автоматически, а не записи, прочитанной с помощью инструкции getline. Последняя инструкция выводит строку и значение NR, номер текущей записи. Несмотря на то что getline не изменяет значения переменной $0, когда она считывает данные в переменную, gawk увеличивает значение NR на единицу.

```
$ cat g5
        # вывод всех строк за исключением прочитанных с помощью getline
        {print "строка №", NR, $0}
# если строка начинается с "b", она обрабатывается особым образом
```

```
/^b/    {
        # использование getline для чтения следующей строки в переменную hold
        getline hold
        # вывод значения hold
        print "пропуск этой строки:", hold
        # когда getline читает данные в переменную, $0 это не касается
        # $1 содержит свое прежнее значение
        print "предыдущая строка начиналась с:", $1
        }
        {
        print ">>>> завершение обработки строки №", NR
        print ""
        }
```

```
$ gawk -f g5 < alpha
строка № 1 aaaaaaaaa
>>>> завершение обработки строки № 1
строка № 2 bbbbbbbbb
пропуск этой строки: ccccccccc
предыдущая строка начиналась с: bbbbbbbbb
>>>> завершение обработки строки № 3
строка № 4 ddddddddd
>>>> завершение обработки строки № # 4
```

Сопроцесс: Двунаправленный ввод-вывод

Сопроцесс — это процесс, работающий параллельно другому процессу. Начиная с версии 3.1, gawk может вызывать сопроцесс для обмена информацией непосредственно с фоновым процессом. Сопроцесс можно использовать для работы с клиент-серверной средой, для клиент-серверной настройки SQL или для обмена данными с удаленной системой по сети. В синтаксисе gawk сопроцесс идентифицируется путем установки перед именем программы, запускающей фоновый процесс, оператора |&.

СОПРОЦЕССЫ ПОДДЕРЖИВАЮТСЯ ТОЛЬКО В GAWK

Утилиты awk и mawk не поддерживают сопроцессы. Они поддерживаются только в gawk.

Команда сопроцесса должна быть фильтром (то есть она должна читать данные из стандартного ввода и записывать их в стандартный вывод) и должна отправлять свой вывод, как только у нее будет целая строка, а не накапливать строки для их последующего вывода. Когда команда вызывается как сопроцесс, она осуществляет связь с gawk-программой через двунаправленный канал, позволяя читать данные из сопроцесса и записывать их в него.

to_upper

При автономном использовании утилита tr не отправляет свой вывод после каждой строки. Сценарий оболочки to_upper является контейнером для утилиты tr, осуществляющим отправку своего вывода; этот фильтр может быть запущен в качестве сопроцесса. Для каждой считанной строки сценарий to_upper записывает строку, переводя ее символы в верхний регистр, в стандартный вывод. Если нужно, чтобы программа to_upper выводила отладочную информацию, следует удалить знак решетки # перед командой set -x.

```
$ cat to_upper
#!/bin/bash
#set -x
while read arg
do
        echo "$arg" | tr '[a-z]' '[A-Z]'
done
```

```
$ echo abcdef | ./to_upper
ABCDEF
```

Программа g6 вызывает to_upper в качестве сопроцесса. Эта gawk-программа читает данные из стандартного ввода или из файла, указанного в командной строке, переводит символы ввода в верхний регистр и записывает преобразованные данные в стандартный вывод.

```
$ cat g6
        {
        print $0 |& "to_upper"
        "to_upper" |& getline hold
        print hold
        }

$ gawk -f g6 < alpha
AAAAAAAAA
BBBBBBBBB
CCCCCCCCC
DDDDDDDDD
```

В программе g6 используется одна составная инструкция, заключенная в кавычки, которая состоит из трех инструкций. Поскольку *шаблон* отсутствует, gawk выполняет составную инструкцию для каждой строки ввода.

В первой инструкции, `print $0`, текущая запись отправляется на стандартный вывод. Оператор `|&` перенаправляет стандартный вывод в программу по имени to_upper, которая запускается в качестве сопроцесса. Имя этой программы нужно взять в кавычки. Вторая инструкция перенаправляет стандартный вывод из to_upper инструкции getline, которая копирует свой стандартный ввод в переменную по имени `hold`. Третья инструкция, `print hold`, отправляет содержимое переменной `hold` на стандартный вывод.

Получение ввода из сети

Основываясь на понятии сопроцесса, gawk может вести обмен информацией с процессом на другой системе через сетевое IP-соединение. При указании одного из специальных имен файлов, которые начинаются со строки `/inet/`, gawk обрабатывает запрос, используя сетевое соединение. Для специальных имен файлов используется следующий формат:

/inet/протокол/локальный_порт/удаленный_хост/удаленный_порт

где в качестве *протокола* обычно указывается tcp, но может использоваться и udp, в качестве *локального_порта* указывается 0 (нуль), если нужно, чтобы порт был выбран самой утилитой gawk (или указывается номер требуемого порта), в качестве *удаленного_хоста* указывается IP-адрес или полное доменное имя удаленного хоста и в качестве *удаленного_порта* указывается номер порта на удаленном хосте. При указании *локального_порта* и *удаленного_порта* вместо номеров портов можно использовать имя службы, например http или ftp.

Программа g7 читает файл rfc-retrieval.txt с сервера, который находится по адресу www.rfc-editor.org. На сайте www.rfc-editor.org файл находится в каталоге /rfc/rfc-retrieval.txt. Первая инструкция в программе g7 присваивает специальное имя файла переменной `server`. В имени файла указывается TCP-соединение, позволяющее локальной системе выбрать соответствующий порт и подключиться к www.rfc-editor.org с использованием порта 80. Для указания стандартного HTTP-порта вместо порта 80 можно использовать аббревиатуру http.

Вторая инструкция для отправки GET-запроса на удаленный сервер использует сопроцесс. Этот запрос включает в себя путевое имя файла, запрашиваемого gawk. Цикл while использует сопроцесс для перенаправления строк с сервера инструкции getline. Поскольку этой инструкции не указано в виде аргумента имя переменной, она сохраняет свой ввод в буфере текущей записи $0. Последняя инструкция print отправляет каждую запись на стандартный вывод. Проведите

с этим сценарием эксперимент, заменив последнюю инструкцию print инструкцией утилиты gawk, которая осуществляет обработку файла.

```
$ cat g7
BEGIN           {
        # присваивание переменной по имени server
        # специального сетевого имени файла
        server = "/inet/tcp/0/www.rfc-editor.org/80"

        # использование сопроцесса для отправки GET-запроса на удаленный сервер
        print "GET /rfc/rfc-retrieval.txt" |& server

        # цикл while использует сопроцесс для перенаправления
        # вывода с сервера инструкции getline
        while (server |& getline)
            print $0
        }

$ gawk -f g7
                    Where and how to get new RFCs
                    -----------------------------

RFCs may be obtained via FTP or HTTP or email from many RFC repositories.
The official repository for RFCs is:
        http://www.rfc-editor.org/
...
```

Заключение

AWK является языком сканирования и обработки данных с помощью шаблонов, позволяющим вести поиск в одном или нескольких файлах записей (в качестве которых обычно выступают строки), соответствующих указанным шаблонам. Программа на этом языке обрабатывает строки, выполняя, к примеру, такие действия, как отправка записей на стандартный вывод или увеличение значения счетчика на единицу при каждом обнаружении соответствия шаблону. AWK имеет несколько реализаций, среди которых awk, gawk и mawk.

AWK-программа состоит из одной или нескольких строк, содержащих *шаблон* и (или) *действие*, для которых используется следующий формат:

шаблон {действие}

Шаблон выбирает строки из ввода. AWK-программа выполняет *действие* над всеми строками, выбранными с помощью шаблона. Если *программная* строка не содержит *шаблон*, AWK выбирает все строки ввода. Если программная строка не содержит *действие*, AWK копирует выбранные строки на стандартный вывод.

AWK-программа может использовать переменные, функции, арифметические операторы, ассоциативные массивы, управляющие инструкции и инструкцию printf, позаимствованную у языка C. В расширенном программировании на AWK для более точной настройки ввода используется инструкция getline, а также используются сопроцессы (поддерживаемые только в gawk), которые позволяют gawk вести обмен

данными с другими программами, и сетевые подключения для обмена данными по сети с программами, запущенными на удаленных системах (только в gawk).

Упражнения

1. Напишите AWK-программу, нумерующую каждую строку, получаемую из файла, и отправляющую данные на стандартный вывод.
2. Напишите AWK-программу, которая выводит количество символов в первом поле, за которым следует содержимое первого поля, и отправляет данные на стандартный вывод.
3. Напишите AWK-программу, которая использует файл cars (см. подраздел «Файл данных cars» раздела «Инструкция continue» на с. 562), выводит данные о всех автомобилях, цена которых превышает $5000, и отправляет эти данные на стандартный вывод.
4. Используйте AWK для определения, сколько строк в файле /usr/share/dict/words содержат подстроку `abul`. Проверьте полученный ответ с помощью команды grep.

Упражнения повышенной сложности

1. Проведите эксперимент с утилитой pgawk (доступной только вместе с gawk). Что она делает? Как ее можно использовать?
2. Напишите gawk-программу (не предназначенную для awk или mawk) по имени net_list, которая читает данные из файла rfc-retrieval.txt на сайте www.rfc-editor.org (см. раздел «Получение ввода из сети» на с. 582) и выводит последнее слово каждой строки с буквами, переведенными в верхний регистр.
3. Доработайте программу net_list, разработанную по заданию упражнения 6, под использование в качестве сопроцесса программы to_upper (см. подраздел «to_upper» раздела «Примеры» на с. 581) для вывода списка автомобилей, в котором буквами в верхнем регистре будет обозначен только производитель. Модель и последующие поля каждой строки должны появляться в том виде, в котором они сохранены в файле cars.
4. Как можно получить gawk (но не awk или mawk) с четко читаемым форматом, то есть как привести распечатку файла gawk-программы к структурированному виду? (*Подсказка.* Обратитесь к man-странице, посвященной gawk.)

Глава 13. Редактор sed

Утилита sed (stream editor — потоковый редактор) является пакетным (неинтерактивным) редактором. Она преобразует вводимый поток, который может поступать из файла или из стандартного ввода. Эта утилита часто используется в канале в качестве фильтра. Поскольку она осуществляет только один проход вводимой информации, утилита sed является более эффективным средством по сравнению с таким интерактивным редактором, как ed. Со многими дистрибутивами Linux поставляется утилита GNU sed, а в системе Mac OS X предоставляется утилита BSD sed. Материалы настоящей главы могут применяться к обеим версиям этой утилиты.

Синтаксис

Для командной строки sed используется следующий синтаксис:

sed [–n] программа [список_файлов]
sed [–n] –f файл_программы [список_файлов]

Утилита sed получает свой ввод из файлов, указанных в командной строке, или из стандартного ввода. Вывод из sed поступает на стандартный вывод.

Аргументы

Программа — это программа sed, являющаяся частью командной строки. Первый формат позволяет создавать простые, короткие sed-программы без создания отдельного файла для хранения sed-программы. *Файл_программы* во втором формате — это путевое имя файла, в котором содержится sed-программа (см. раздел «Основы редактора» на с. 586). *Список_файлов* содержит путевые имена обычных файлов, обрабатываемых sed; это файлы, которые служат для редактора вводом. Если *список_файлов* не указан, sed использует стандартный ввод.

Ключи

Ключи, перед которыми стоит двойной дефис (--), работают только под управлением Linux (в GNU sed). Имена ключей, состоящие из одной буквы, перед которой ставится одиночный дефис, работают как под Linux (GNU sed), так и под OS X (BSD sed).

`--file` *файл_программы*
`-f` *файл_программы*

Заставляет sed читать его программу из файла по имени *файл_программы*, а не из командной строки. Этот ключ может использоваться в командной строке более одного раза.

`--help`

Вызывает вывод краткой справки по использованию sed. (Linux)

`--in-place[=`*суффикс*`]`
`-i[`*суффикс*`]`

Редактирует файлы на месте. Без этого ключа **sed** отправляет выводимую информацию на стандартный вывод. При использовании этого ключа **sed** заменяет обрабатываемый им файл своим выводом. При указании *суффикса* **sed** создает резервную копию исходного файла. У резервной копии сохраняется исходное имя файла с добавлением к нему *суффикса*. Если нужно, чтобы между исходным именем файла и суффиксом появилась точка, ее нужно включить в *суффикс*.

`--quiet` или `-n`
`--silent`

Заставляет **sed** не копировать строки на стандартный вывод, за исключением тех строк, в отношении которых применяется инструкция Print (p) или соответствующий флаг.

Основы редактора

Программа **sed** состоит из одной или нескольких строк, имеющих следующий синтаксис:

[адрес [,адрес]] инструкция [список_аргументов]

Адреса являются необязательной частью синтаксиса. Если *адрес* опустить, **sed** обрабатывает все строки ввода. *Инструкция* — это инструкция редактирования, изменяющая текст. *Адреса* выбирают строку (строки), с которой работает часть команды, относящаяся к *инструкции*. Количество и разновидности аргументов в *списке_аргументов* зависят от *инструкции*. Если в одну командную строку нужно поместить несколько команд **sed**, их следует отделить друг от друга точками с запятой (;).

Утилита **sed** обрабатывает ввод следующим образом:

1. Читает одну строку ввода из *списка_файлов* или из стандартного ввода.
2. Читает первую *инструкцию* из *программы* или из *файла_программы*. Если строка ввода выбирается *адресом* (или адресами), обрабатывает строку ввода согласно *инструкции*.
3. Читает следующую инструкцию из *программы* или из *файла_программы*. Если строка ввода выбирается *адресом* (или адресами), обрабатывает строку ввода (возможно, уже измененную предыдущей *инструкцией*) согласно *инструкции*.
4. Повторяет шаг 3 до тех пор, пока не будут выполнены все инструкции в *программе* или в *файле_программы*.
5. Начинает с шага 1, если есть еще одна строка ввода; если такой строки нет, **sed** завершает работу.

Адреса

Номер строки является адресом, выбирающим строку. В качестве особого случая, номер, обозначенный символом $, является представителем последней строки ввода.

Регулярное выражение является адресом, выбирающим строки, содержащие соответствующие этому выражению строковые значения. Хотя в качестве ограничителей регулярных выражений чаще всего используются слэши, sed позволяет использовать для этих целей любые символы, кроме обратного слэша и разделителя строк.

Кроме того, уже отмечалось, что перед инструкцией может не быть вообще никакого адреса, или же быть один или два адреса (в виде либо номеров строк, либо регулярных выражений). Если адрес не будет указан, sed выбирает все строки, заставляя инструкцию обрабатывать каждую строку ввода. Указание одного адреса заставляет инструкцию обрабатывать каждую строку ввода, выбранную адресом. Указание двух адресов заставляет инструкцию обрабатывать группу строк. В этом случае первый адрес выбирает первую строку в первой группе. Второй адрес выбирает следующую за первой строку, соответствующую этому адресу; и эта строка становится последней строкой первой группы. Если соответствий для второго адреса найдено не будет, то он укажет на конец файла. После выбора последней строки в группе sed начинает новый процесс выбора с поиска следующей строки, соответствующей первому адресу. Эта строка становится первой строкой следующей группы. Утилита sed продолжает этот процесс, пока не будет пройден весь файл.

Инструкции

Область шаблона — Утилита sed использует два буфера. Следующие команды работают с *Областью шаблона*, в котором изначально хранится строка ввода, только что прочитанная sed. Другой буфер, *Область хранения*, рассмотрен далее в соответствующем разделе.

a — (append — добавить). Инструкция Добавить добавляет к текущей выбранной строке одну или несколько строк. Если перед инструкцией Добавить указать два адреса, она осуществляет добавление к каждой строке, выбранной адресами. Если перед инструкцией Добавить не указать адрес, добавление будет выполнено в отношении каждой строки ввода. Инструкция Добавить использует следующий формат:

```
[адрес [,адрес]] a\
текст \
текст \
...
текст
```

Каждую строку добавляемого текста, кроме последней, нужно заканчивать символом обратного слэша, который цитирует следующий за ним символ разделителя строк. Добавляемый текст завершается строкой, не заканчивающейся символом обратного слэша. Утилита sed всегда выводит добавляемый текст, независимо от того, используется или нет ключ -n в командной строке. Она выводит текст даже в том случае, если строка, к которой он добавляется, удалена.

c — (change — изменить). Инструкция Изменить похожа на инструкции Добавить и Вставить, за исключением того, что она изменяет выбранные строки, делая

их содержимым новый текст. Когда указывается диапазон адресов, инструкция Изменить заменяет строки этого диапазона одним экземпляром нового текста.

d (delete — удалить). Инструкция Удалить заставляет sed не выводить выбранные им строки и не завершать обработку строк. После того как утилита sed выполнит инструкцию Удалить, она читает следующую строку ввода, а затем начинает все заново, с первой инструкции из *программы* или из *файла_программы*.

i (insert — вставить). Инструкция Вставить идентична инструкции Добавить, за исключением того, что она помещает новый текст перед выбранной строкой.

N (next, в данном случае, следующая без записи). Инструкция Следующая (N) читает следующую строку ввода и добавляет ее к текущей строке. Исходная и новая строка отделяются друг от друга вставленным разделителем строк. Команду N можно использовать для удаления разделителей строк из файла. См. пример в подразделе «Следующая (N)» раздела «Примеры» на с. 595.

n (next — следующая). Инструкция Следующая (n) выводит текущую выбранную строку, если это соответствует установленному режиму работы, читает следующую строку ввода и приступает к обработке новой строки, используя следующую инструкцию из *программы* или из *файла_программы*.

p (print — вывести). Инструкция Вывести записывает выбранные строки на стандартный вывод, делая это сразу же, без применения к этим строкам всех следующих инструкций. Эта инструкция отменяет действие ключа командной строки -n.

q (quit — завершить). Инструкция Завершить заставляет sed немедленно завершить работу.

r *файл* (read — прочитать). Инструкция Прочитать читает содержимое указанного файла и добавляет его к выбранной строке. После инструкции Прочитать должен следовать одиночный пробел и имя файла ввода.

s (substitute — Подставить). Имеющаяся в sed инструкция Подставить аналогична одноименной инструкции редактора vim (см. раздел «Подстановка одной строки вместо другой» на с. 196). Для нее используется следующий формат:

[адрес[,адрес]] s/шаблон/строка_замены /[g][p][w файл]

Шаблон — это регулярное выражение, которое традиционно ограничено слэшем (/), но можно использовать и любой другой символ, кроме пробела и разделителя строк. *Строка_замены* следует сразу же за вторым ограничителем и должна быть завершена таким же ограничителем. Завершающий (третий) ограничитель является обязательным элементом. *Строка_замены* может содержать символ амперсанда (&), который sed заменяет строковым значением, соответствующим *шаблону*. Пока не будет использован флаг g, инструкция Подставить заменяет только первое соответствие шаблону в каждой выбранной строке.

Флаг g (global — глобальный режим) заставляет инструкцию Подставить заменить все неперекрывающиеся строковые значения, соответствующие шаблону в выбранных строках.

Флаг p (print — вывести) заставляет sed отправить все строки, в которых он осуществил подстановку, на стандартный вывод. Этот флаг отменяет действие ключа командной строки -n.

Флаг w (write — записать) аналогичен флагу p, но отправляет вывод в файл по имени *файл*. За флагом w должен следовать одиночный пробел и имя файла вывода.

w *файл* (write — записать). Эта инструкция аналогична инструкции Вывести, за исключением того, что она отправляет вывод в файл по имени *файл*. За инструкцией Записать должен следовать одиночный пробел и имя файла вывода.

Управляющие структуры

! (НЕ). Заставляет sed применить следующую инструкцию, находящуюся в той же строке, к каждой строке, не выбранной адресной частью инструкции. Например, 3!d удаляет все строки, кроме строки 3, а $!p выводит все строки, кроме последней.

{} (группировка инструкций). Когда группа инструкций заключается в фигурные скобки, одиночный адрес или пара адресов выбирают строки, которые обрабатываются группой инструкций. Несколько команд, находящихся в одной строке, отделяются друг от друга точками с запятой (;).

Инструкции ветвления Инструкции ветвления названы в info-странице, посвященной GNU sed, не иначе как «Commands for sed gurus» («Команды для sed-гуру») и предлагают в случае возникновения потребностей в подобных командах рассмотреть более приемлемый вариант написания программы на языке awk или Perl.

: *метка* Является идентификатором места внутри sed-программы. *Метка* используется как цель перехода для инструкций ветвления b или t.

b [*метка*] Безусловная передача управления на *метку*. Без использования *метки* — пропуск всех оставшихся инструкций для текущей строки и чтение следующей строки ввода.

t [*метка*] Передача управления на *метку* только в том случае, если инструкция Подставить была успешно выполнена сразу же после чтения строки, прочитанной непосредственно перед выполнением этой инструкции (условное ветвление). Без использования *метки* — пропуск всех оставшихся инструкций для текущей строки ввода и чтение следующей строки ввода.

Область хранения

Все ранее рассмотренные команды работают с *Областью шаблона*, буфером, в котором изначально содержится строка ввода, только что прочитанная утилитой sed. *Область хранения* может содержать данные только в процессе работы с данными, находящимися в *Области шаблона*; эта область играет роль временного буфера. Пока данные не записаны в *Область хранения*, этот буфер остается пустым. В этом разделе рассматриваются команды, перемещающие данные между *Областью шаблона* и *Областью хранения*.

g Копирование содержимого *Области хранения* в *Область шаблона*. Исходное содержимое *Области шаблона* утрачивается.

G Добавление разделителя строк и содержимого *Области хранения* к *Области шаблона*.

h Копирование содержимого *Области шаблона* в *Область хранения*. Исходное содержимое *Области хранения* утрачивается.

H Добавление разделителя строк и содержимого *Области шаблона* к *Области хранения*.

x Обмен содержимым между *Областью шаблона* и *Областью хранения*.

Примеры

файл данных lines

Следующие примеры используют для ввода данных файл lines:

```
$ cat lines
Line one.                          Строка один.
The second line.                   Вторая строка.
The third.                         Третья.
This is line four.                 Это строка четыре.
Five.                              Пять.
This is the sixth sentence.        Это шестое предложение.
This is line seven.                Это строка семь.
Eighth and last.                   Восьмая и последняя.
```

Пока не будет дана соответствующая инструкция, sed отправляет все строки — выбранные или нет — на стандартный вывод. Когда в командной строке используется ключ -n, sed отправляет на стандартный вывод только вполне определенные строки, например те, которые были выбраны инструкцией Вывести (p).

Следующая командная строка выводит все строки файла, в которых содержится слово line (с буквами в нижнем регистре). Кроме этого, поскольку ключ -n отсутствует, sed выводит все строки ввода. В результате sed выводит строки, которые содержат слово line, дважды.

```
$ sed '/line/ p' lines
Line one.
The second line.
The second line.
The third.
This is line four.
This is line four.
Five.
This is the sixth sentence.
This is line seven.
This is line seven.
Eighth and last.
```

В предыдущей команде используется адрес /line/, регулярное выражение, представленное простой строкой. Утилита sed выбирает каждую строку, которая содержит

соответствие этому шаблону. Инструкция Вывести (p) выводит каждую выбранную строку.

В следующей команде используется ключ -n, поэтому sed выводит только выбранные строки:

```
$ sed -n '/line/ p' lines
The second line.
This is line four.
This is line seven.
```

В следующем примере sed выводит часть файла на основе указанных номеров строк. Инструкция Вывести выбирает и выводит строки с 3 по 6.

```
$ sed -n '3,6 p' lines
The third.
This is line four.
Five.
This is the sixth sentence.
```

В следующей командной строке используется инструкция Завершить, заставляющая sed вывести только начало файла. В данном случае sed выводит первые пять строк, точно так же, как это сделала бы команда head -5.

```
$ sed '5 q' lines
Line one.
The second line.
The third.
This is line four.
Five.
```

файл_программы

Когда нужно дать sed более сложные или длинные инструкции, можно воспользоваться *файлом_программы*. Программа print3_6 выполняет ту же функцию, что и командная строка в предпоследнем примере. Ключ -f заставляет sed читать свою программу из файла, чье имя указано сразу же за этим ключом.

```
$ cat print3_6
3,6 p

$ sed -n -f print3_6 lines
The third.
This is line four.
Five.
This is the sixth sentence.
```

Добавить

Следующая программа выбирает строку 2 и использует инструкцию Добавить, чтобы добавить разделитель строк и текст AFTER. ко второй строке. Поскольку в командной строке нет ключа -n, sed копирует все строки из файла ввода lines.

```
$ cat append_demo
2 a\
AFTER.

$ sed -f append_demo lines
Line one.
The second line.
```

```
AFTER.
The third.
This is line four.
Five.
This is the sixth sentence.
This is line seven.
Eighth and last.
```

Вставить Программа insert_demo выбирает все строки, в которых содержится строка This, и вставляет разделитель строк и текст BEFORE. перед выбранными строками.

```
$ cat insert_demo
/This/ i\
BEFORE.

$ sed -f insert_demo lines
Line one.
The second line.
The third.
BEFORE.
This is line four.
Five.
BEFORE.
This is the sixth sentence.
BEFORE.
This is line seven.
Eighth and last.
```

Изменить В следующем примере показана работа инструкции Изменить с диапазоном адресов. При указании инструкции Изменить диапазона строк она не изменяет каждую строку внутри диапазона, а меняет блок, вставляя вместо него один экземпляр нового текста.

```
$ cat change_demo
2,4 c\
SED WILL INSERT THESE\                  SED ВСТАВИТ ЭТИ\
THREE LINES IN PLACE\                   ТРИ СТРОКИ ВМЕСТО\
OF THE SELECTED LINES.                  ВЫБРАННЫХ СТРОК.

$ sed -f change_demo lines
Line one.
SED WILL INSERT THESE
THREE LINES IN PLACE
OF THE SELECTED LINES.
Five.
This is the sixth sentence.
This is line seven.
Eighth and last.
```

Подставить В следующем примере показывается работа инструкции Подставить. Утилита sed выбирает все строки, поскольку у инструкции отсутствует адрес. В каждой строке программа subs_demo заменяет первый экземпляр слова

line словом sentence. Флаг p приводит к выводу каждой строки, в которой произошла подстановка. Командная строка вызывает sed с ключом -n, поэтому sed выводит только те строки, которые конкретно указаны программой.

```
$ cat subs_demo
s/line/sentence/p

$ sed -n -f subs_demo lines
The second sentence.
This is sentence four.
This is sentence seven.
```

Следующий пример похож на предыдущий, за исключением флага w и имени файла (temp) в конце инструкции Подставить, которые заставляют sed создать файл по имени temp. В командной строке отсутствует ключ -n, поэтому она выводит все строки, а кроме этого, записывает измененные строки в файл temp. Утилита cat выводит содержимое файла temp. Слово Line (начинающееся с заглавной буквы L) изменению не подвергается.

```
$ cat write_demo1
s/line/sentence/w temp

$ sed -f write_demo1 lines
Line one.
The second sentence.
The third.
This is sentence four.
Five.
This is the sixth sentence.
This is sentence seven.
Eighth and last.

$ cat temp
The second sentence.
This is sentence four.
This is sentence seven.
```

Следующий сценарий оболочки bash изменяет в группе файлов все появления слова REPORT, превращая их в report, все слова FILE превращает в file, а слова PROCESS — в process. Поскольку это сценарий оболочки, а не программный файл sed, у вас должны быть права на чтение файла sub и на его выполнение в качестве команды (см. раздел «chmod: Превращение файла в исполняемый» на с. 292). Структура for (см. раздел «Управляющая структура for» на с. 430) осуществляет последовательный перебор файлов, указанных в командной строке. При обработке каждого файла, перед его обработкой с помощью утилиты sed, сценарий выводит имя этого файла. Данная программа использует встроенные sed-команды, охватывающие сразу несколько строк. Поскольку разделители строк между командами процитированы (взяты в одиночные кавычки), sed воспринимает несколько команд как одну расширенную командную строку (внутри сценария оболочки). Каждая инструкция Подставить включает флаг g (global — глобальный режим), чтобы позаботиться о случае многократного присутствия строкового значения в строке.

```
$ cat sub
for file
```

```
do
        echo $file
        mv $file $$.subhld
        sed 's/REPORT/report/g
             s/FILE/file/g
             s/PROCESS/process/g' $$.subhld > $file
done
rm $$.subhld

$ sub file1 file2 file3
file1
file2
file3
```

В следующем примере инструкция Записать копирует часть файла в другой файл (temp2). Номера строк 2 и 4, разделенные запятой, выбирают диапазон строк, копируемый утилитой sed. Эта программа не вносит изменения в строки.

```
$ cat write_demo2
2,4 w temp2

$ sed -n -f write_demo2 lines

$ cat temp2
The second line.
The third.
This is line four.
```

Программа write_demo3 похожа на программу write_demo2, но перед инструкцией Записать стоит оператор НЕ (!), заставляющий sed записывать в файл те строки, которые не выбраны с помощью адреса.

```
$ cat write_demo3
2,4 !w temp3

$ sed -n -f write_demo3 lines

$ cat temp3
Line one.
Five.
This is the sixth sentence.
This is line seven.
Eighth and last.
```

Следующая (n)

В следующем примере показана работа инструкции Следующая (n). При обработке выбранной строки (строки 3) sed тут же приступает к обработке следующей строки, не выводя строку 3.

```
$ cat next_demo1
3 n
p

$ sed -n -f next_demo1 lines
Line one.
The second line.
```

```
This is line four.
Five.
This is the sixth sentence.
This is line seven.
Eighth and last.
```

В следующем примере используется текстовый адрес. Строка the содержится в шестой строке, поэтому инструкция Следующая (n) заставляет sed не выводить эту строку.

```
$ cat next_demo2
/the/ n
p

$ sed -n -f next_demo2 lines
Line one.
The second line.
The third.
This is line four.
Five.
This is line seven.
Eighth and last.
```

Следующая (N)

Следующий пример похож на предыдущий, за исключением того, что вместо инструкции Следующая с буквой в нижнем регистре (n) используется инструкция Следующая с буквой в верхнем регистре (N). Данная инструкция Следующая (N) добавляет следующую строку к той строке, которая содержит слово the. В файле lines sed добавляет строку 7 к строке 6, вставляет между ними разделитель строк. Команда Подставить заменяет вставленный разделитель строк пробелом. Команда Подставить не оказывает влияние на содержимое других строк, поскольку в них не содержатся вставленные разделители строк, вернее, эти строки заканчиваются разделителями строк.

```
$ cat Next_demo3
/the/ N
s/\n/ /
p

$ sed -n -f Next_demo3 lines
Line one.
The second line.
The third.
This is line four.
Five.
This is the sixth sentence. This is line seven.
Eighth and last.
```

В следующих нескольких примерах для демонстрации совместной работы инструкций утилиты sed используется файл compound.in.

```
$ cat compound.in
1. The words on this page…          Слова на этой странице…
2. The words on this page…
3. The words on this page…
4. The words on this page…
```

Следующий пример осуществляет подстановку вместо строкового значения words слова text в строках 1–3 и вместо строкового значения text слова TEXT в строках 2–4. В примере также выбирается и удаляется строка 3. В результате появляется слово text в строке 1, слово TEXT в строке 2, отсутствует строка 3 и остается слово words в строке 4. Утилита sed осуществляет две подстановки в строках 2 и 3: text вместо words, и TEXT вместо text. Затем sed удаляет строку 3.

```
$ cat compound
1,3 s/words/text/
2,4 s/text/TEXT/
3 d

$ sed -f compound compound.in
1. The text on this page…
2. The TEXT on this page…
4. The words on this page…
```

Порядок следования инструкций в sed-программе играет очень важную роль. В предыдущем примере ко второй строке применяются обе инструкции Подставить, и поэтому изменение порядка осуществления подстановки приводит к изменению конечного результата.

```
$ cat compound2
2,4 s/text/TEXT/
1,3 s/words/text/
3 d

$ sed -f compound2 compound.in
1. The text on this page…
2. The text on this page…
4. The words on this page…
```

В следующем примере программа compound3 добавляет ко второй строке две строки. Утилита sed выводит сразу все строки из файла, поскольку в командной строке отсутствует ключ -n. Инструкция Вывести в конце файла программы выводит строку 3 еще раз.

```
$ cat compound3
2 a\
This is line 2a.\                    Это строка 2a
This is line 2b.                     Это строка 2b
3 p

$ sed -f compound3 compound.in
1. The words on this page…
2. The words on this page…
This is line 2a.
This is line 2b.
3. The words on this page…
3. The words on this page…
4. The words on this page…
```

В следующем примере показывается, что sed всегда выводит добавленный текст. В этом примере удаляется строка 2, но инструкция Добавить по-прежнему выводит

две строки, добавленные к строке 2. Добавленные строки выводятся, даже если в командной строке используется ключ -n.

```
$ cat compound4
2 a\
This is line 2a.\
This is line 2b.
2 d

$ sed -f compound4 compound.in
1. The words on this page…
This is line 2a.
This is line 2b.
3. The words on this page…
4. The words on this page…
```

В следующем примере в качестве шаблона используется регулярное выражение. Это регулярное выражение в следующей инструкции (^.) соответствует любому одинарному символу в начале каждой непустой строки. Строка замены (между вторым и третьим слэшами) содержит эскейп-последовательность, использующую обратный слэш и обозначающую символ табуляции (\t), за которой следует знак амперсанда (&). Этот амперсанд превращается в значение строкового выражения, которому соответствует регулярное выражение.

```
$ sed 's/^./\t&/' lines
        Line one.
        The second line.
        The third.
…
```

Такой вариант подстановки используется для отступа в файле, создающего поле слева.

Для добавления символа табуляции в начало всех строк можно воспользоваться и более простой формой s/^/\t/. Кроме установки символов табуляции в начало строк, содержащих текст, эта инструкция поставит символ табуляции и в начало каждой пустой строки, чего не делает предыдущая команда.

Чтобы предыдущую sed-инструкцию не пришлось заучивать (и перенабирать) всякий раз, когда нужно будет добавить отступы в файл, можно вставить ее в сценарий оболочки. Права на чтение и выполнение файла ind можно получить с помощью утилиты chmod.

```
$ cat ind
sed 's/^./\t&/' $*
$ chmod u+rx ind
$ ind lines
        Line one.
        The second line.
        The third.
…
```

Автономный сценарий

При запуске предыдущего сценария оболочки создаются два процесса: вызывается оболочка, которая, в свою очередь, вызывает утилиту sed. Издержек, связанных с созданием процесса оболочки, можно избежать, если вставить

в начало сценария строку #!/bin/sed -f (см. раздел «#! — способ указания оболочки» на с. 294), которая приведет к непосредственному запуску утилиты sed. При этом у вас должны быть права на чтение и выполнение файла, в котором хранится сценарий.

```
$ cat ind2
#!/bin/sed -f
s/^./\t&/
```

В следующей sed-программе регулярное выражение (два пробела, за которыми следуют символы *$) соответствует одному или нескольким пробелам в конце строки. Эта программа удаляет замыкающие пробелы в конце строк, что может пригодиться при очистке файлов, созданных с помощью редактора vim.

```
$ cat cleanup
sed 's/ *$//' $*
```

Сценарий cleanup2 запускает такую же sed-команду, что и программа cleanup, но при этом работает в автономном режиме, то есть вызывает sed напрямую, без посредничества оболочки.

```
$ cat cleanup2
#!/bin/sed -f
s/ *$//
```

Область хранения

Следующая sed-программа использует *Область хранения*, чтобы поменять местами две строки файла.

```
$ cat s1
h # Копирование Области шаблона (только что прочитанной строки) в Область хранения.
n # Чтение следующей строки ввода в Область шаблона.
p # Вывод содержимого Области шаблона.
g # Копирование Области хранения в Область шаблона.
p # Вывод содержимого Области шаблона (теперь уже с предыдущей строкой).

$ sed -nf s1 lines
The second line.
Line one.
This is line four.
The third.
This is the sixth sentence.
Five.
Eighth and last.
This is line seven.
```

Команды в программе s1 осуществляют попарную обработку строк ввода. Эта программа читает и сохраняет строку, потом читает следующую строку и выводит ее на экран, а затем извлекает и выводит на экран сохраненную строку. После обработки пары строк программа начинает все сначала, используя следующую пару строк.

Следующая sed-программа добавляет пустую строку после каждой строки файла ввода (то есть удваивает интервалы между строками файла).

```
$ sed 'G' lines
Line one.

The second line.
```

```
The third.

This is line four.

$
```

Инструкция G добавляет к *Области шаблона* разделитель строк и содержимое *Области хранения*. Пока в *Область хранения* не будут помещены какие-нибудь данные, она будет оставаться пустой. Поэтому инструкция G добавляет разделитель строк к каждой строке ввода еще до того, как sed выводит строку (строки) из *Области шаблона*.

Программа s2 меняет порядок следования строк файла на обратный, точно так же, как это делает утилита tac.

```
$ cat s2
2,$G  # Для всех строк, кроме первой, добавление к Области шаблона
      # разделителя строк и содержимого Области хранения.
h     # Копирование Области шаблона в Область хранения.
$!d   # Удаление, касающееся всех строк, за исключением последней строки.

$ sed -f s2 lines
Eighth and last.
This is line seven.
This is the sixth sentence.
Five.
This is line four.
The third.
The second line.
Line one.
```

Эта программа состоит из трех команд: 2,$G, h и $!d. Чтобы разобраться в работе сценария, важно понимать, как работает адрес последней команды: знак $ является адресом последней строки ввода, а команда ! инвертирует адрес. В результате получается адрес, выбирающий все строки, за исключением последней строки ввода. Точно так же можно заменить первую команду командой 1!G: она будет выбирать для обработки все строки, за исключением первой, и конечный результат будет таким же.

При обработке программой s2 строк файла происходят следующие действия:

1. Утилита sed считывает первую строку ввода (Line one.) в *Область шаблона*.
   1) Команда 2,$G не обрабатывает первую строку ввода. Благодаря своему адресу инструкция G начинает обработку со второй строки.
   2) Команда h копирует Line one. из *Области шаблона* в *Область хранения*.
   3) Команда $!d удаляет содержимое *Области шаблона*. Поскольку в *Области шаблона* ничего не содержится, sed ничего не выводит.
2. Утилита sed считывает вторую строку ввода (The second line.) в *Область шаблона*.
   1) Команда 2,$G добавляет все, что содержится в *Области хранения* (Line one.) в *Область шаблона*. Теперь в *Области шаблона* содержится The second line. разделитель_строкLine one..
   2) Команда h копирует все, что содержится в *Области шаблона*, в *Область хранения*.

3) Команда `$!d` удаляет вторую строку ввода. Поскольку она удалена, sed ее не выводит.

3. Утилита sed считывает третью строку ввода (`The third.`) в *Область шаблона*.
   1) Команда `2,$G` добавляет все, что содержится в *Области хранения* (`The second line.разделитель_строкLine one.`) в *Область шаблона*. Теперь в *Области шаблона* содержится `The third. разделитель_строкThe second line. разделитель_строкLine one..`
   2) Команда `h` копирует все, что содержится в *Области шаблона*, в *Область хранения*.
   3) Команда `$!d` удаляет содержимое *Области шаблона*. Поскольку в *Области шаблона* ничего не остается, sed ничего не выводит.

   ...

4. Утилита sed считывает восьмую (последнюю) строку ввода в *Область шаблона*.
   1) Команда `2,$G` добавляет все, что содержится в *Области хранения*, в *Область шаблона*. Теперь в *Области шаблона* содержатся все строки из файла lines, следующие в обратном порядке.
   2) Команда `h` копирует все, что содержится в *Области шаблона*, в *Область хранения*. Для последней строки ввода этот шаг не нужен, но он не влияет на содержимое данных, выводимых программой.
   3) Команда `$!d` не обрабатывает последнюю строку ввода. Благодаря своему адресу инструкция d не удаляет последнюю сроку.
   4) Утилита sed выводит содержимое *Области шаблона*.

Заключение

Утилита sed (stream editor — потоковый редактор) является пакетным (неинтерактивным) редактором. Она получает ввод из файлов, указанных в командной строке, или же из стандартного ввода. Пока вывод из sed не будет перенаправлен, он будет отправляться на стандартный вывод.

Программа sed состоит из одной или нескольких строк, имеющих следующий синтаксис:

[адрес [,адрес]] инструкция [список_аргументов]

Адреса являются необязательной частью строки. Если *адрес* будет опущен, sed обрабатывает все строки ввода. *Инструкция* представляет собой инструкцию редактирования, вносящую изменения в текст. *Адреса* выбирают строку (строки), с которыми работает та часть команды, которая относится к *инструкции*. Количество и разновидности аргументов в *списке_аргументов* зависят от *инструкции*.

В дополнение к основным инструкциям, sed имеет в своем арсенале ряд мощных, расширенных инструкций. Один из наборов этих инструкций позволяет sed-программам осуществлять временное хранение данных в буфере под названием Область хранения. Другие инструкции предоставляют в sed-программах возможность применения безусловного и условного ветвления.

Упражнения

1. Напишите sed-команду, копирующую файл на стандартный вывод, удаляя при этом все строки, начинающиеся со слова `Today`.
2. Напишите sed-команду, копирующую на стандартный вывод только те строки файла, которые начинаются со слова `Today`.
3. Напишите sed-команду, копирующую файл на стандартный вывод, удаляя при этом все пустые строки (например, строки без символов).
4. Напишите sed-программу по имени ins, которая копирует файл на стандартный вывод, заменяя все встречающиеся слова `cat` на `dog` и предваряя каждую измененную строку строкой предупреждения `следующая строка была изменена:`.
5. Напишите sed-программу по имени div, которая копирует файл на стандартный вывод, копирует первые пять строк в файл по имени first и копирует остальную часть файла в файл по имени last.
6. Напишите sed-команду, копирующую файл на стандартный вывод, заменяя при этом одиночный пробел, являющийся первым символом в строке, нулем (`0`) только в том случае, если за пробелом сразу же следует число (0–9). Например:

```
abc ▸ abc
 abc ▸  abc
 85c ▸ 085c
55b ▸ 55b
 000 ▸ 0000
```

7. Как использовать sed для тройного увеличения интервала между строками в файле (то есть для добавления после каждой строки двух пустых строк)?

Глава 14. Утилита безопасного копирования rsync

Утилита rsync (remote synchronization — удаленная синхронизация) копирует обычный файл или иерархию каталогов на локальной системе или с локальной системы на другую систему в сети или наоборот. По умолчанию для передачи файлов эта утилита использует OpenSSH и такой же механизм аутентификации, как в OpenSSH, поэтому она предоставляет такую же степень безопасности, как и OpenSSH. Если нужно, утилита rsync выводит приглашение на ввод пароля. В качестве альтернативы можно использовать агентом передачи утилиту rsyncd, работающую в фоновом режиме.

Синтаксис

Для командной строки rsync используется следующий синтаксис:

rsync [ключи] [[пользователь@]с_хоста:] файл-источник [[пользователь@]на_хост:][файл-приемник]

Утилита rsync копирует файлы, включая иерархию каталогов, на локальной системе или между локальной и удаленной системами.

Аргументы

Аргумент *с_хоста* является именем системы, с которой копируются файлы, а аргумент *на_хост* является именем системы, на которую копируются файлы. Если хост не указан, rsync предполагает, что работает с локальной системой. В качестве *пользователя* на обеих системах по умолчанию устанавливается имя того пользователя, который дал команду на локальной системе; с помощью аргумента *пользователь*@ можно указать другое имя пользователя. В отличие от утилиты scp rsync не позволяет осуществлять копирование между двумя удаленными системами.

Файл-источник — это обычный файл или файл каталога, который копируется, а *файл-приемник* — это получаемая копия. Указывать файлы можно с помощью как относительных, так и абсолютных имен файлов. На локальной системе относительные имена определяются относительно рабочего каталога, а на удаленной системе они определяются относительно указанного или подразумеваемого личного каталога пользователя. Когда *файл-источник* является каталогом, для копирования его содержимого следует использовать ключ `--recursive` или ключ `--archive`. Когда каталогом является *файл-приемник*, то каждый исходный файл сохраняет свое простое имя файла. Если *файл-источник* является простым файлом, то *файл-приемник* можно опустить; скопированный файл будет иметь такое же имя, как и *файл-источник* (это свойство можно использовать только при копировании на удаленную машину или с нее).

ЗАМЫКАЮЩИЙ СЛЭШ (/) В ФАЙЛЕ-ИСТОЧНИКЕ ИГРАЕТ ВАЖНУЮ РОЛЬ

Когда файл-источник является каталогом, замыкающий слэш в аргументе файл-источник заставляет утилиту rsync копировать содержимое каталога. Этот слэш эквивалентен символам /*; он заставляет rsync проигнорировать сам каталог и копировать находящиеся внутри него файлы. Без замыкающего слэша rsync копирует каталог. См. раздел «Использование замыкающего слэша (/) в файле-источнике» на с. 606.

Ключи

ВЕРСИЯ RSUNC, ИСПОЛЬЗУЕМАЯ В MAC OS X, ВОСПРИНИМАЕТ ДЛИННЫЕ КЛЮЧИ

Ключи rsync, перед которыми ставится двойной дефис (– –), работают как под управлением Mac OS X, так и под управлением Linux.

`--acls -A` Сохранение ACL-списков копируемых файлов (см. раздел «Списки управления доступом (ACL-списки)» на с. 121).

`--archive -a` Рекурсивное копирование файлов, включая разыменованные символьные ссылки, файлы устройств и специальные файлы, с сохранением связанных с файлами сведений об их владельцах, групповых и индивидуальных правах доступа и времени последнего изменения. Использование этого ключа равносильно указанию ключей `--devices`, `--specials`, `--group`, `--links`, `--owner`, `--perms`, `--recursive` и `--times`. Этот ключ не включает в себя действие ключей `--acls`, `--hard-links` или `--xattrs`. Если это необходимо, их следует указывать к ключу `-archive` дополнительно. Пример показан в подразделе «--archive» раздела «Использование замыкающего слэша (/) в файле-источнике» на с. 606.

`--backup -b` Переименование файлов, которые в противном случае должны быть удалены или переписаны. По умолчанию утилита rsync переименовывает файлы, добавляя к их существующим именам знак тильды (~). Если нужно чтобы rsync вместо переименования поместила эти файлы в указанный каталог, следует воспользоваться ключом `--backup-dir=`*dir*. См. также ключ `--link-dest=`*dir* на с. 604.

`--backup-dir=`*dir* Использование ключа `--backup` приводит к перемещению файлов, которые в противном случае должны быть удалены или переписаны в каталог *dir*. После перемещения старой версии файла в каталог *dir* новая версия файла из *файла-источника* копируется в *файл-приемник*.

Каталог по имени *dir* находится на той же локальной системе, что и *файл-приемник*. Если *dir* указан с использованием относительного путевого имени, это имя определяется относительно *файла-приемника*.

`--copy-unsafe-links` (частичное разыменование). Для каждого файла, являющегося символьной ссылкой на файл за пределами той иерархии, которой принадлежит *файл-источник*, производится копирование файла, на который указывает ссылка, а не символьной ссылки на него. Без этого ключа утилита rsync копирует все символьные ссылки, даже если она не копирует файл, на который делается ссылка.

`-D` То же самое, что и использование ключей `--devices --specials`.

`--delete` Удаление файлов в *файле-приемнике*, отсутствующих в *файле-источнике*. При использовании данного ключа можно легко удалить все еще нужные файлы (см. врезку-предупреждение в разделе «Удаление файлов» на с. 607).

`--devices` Копирование файлов устройств (только для пользователей, имеющих root-права).

`--dry-run` Запуск rsync без записи на диск. С ключом `--verbose` ключ `--dry-run` сообщает о том, что сделала бы утилита rsync, будь она запущена без него. Этот ключ полезно использовать с ключом `--delete`.

`--group -g` Сохранение групповых связей копируемых файлов.

`--hard-links -H` Сохранение жестких ссылок копируемых файлов.

`--links -l` (буква «l» в нижнем регистре; без разыменования). Для каждого файла, являющегося символьной ссылкой, копирование символьной ссылки, а не файла, на которую указывает ссылка, даже если файл, на который указывает ссылка, не присутствует в *файле-источнике*.

`--link-dest=`*dir* Если утилита rsync копирует файл как обычно, то есть если файл существует в *файле-источнике*, но его нет в *файле-приемнике*, она ищет в каталоге по имени *dir* файл с точно таким же именем. Если она находит в *dir* точно такую же копию файла, rsync создает из файла в *dir* жесткую ссылку на *файл-приемник*. Если точная копия не будет найдена, *rsync* копирует файл в *файл-приемник*. См. подраздел «– –link-dest=*dir*» в разделе «Создание резервных копий» на с. 604.

Каталог *dir* находится в той же системе, что и *файл-приемник*. Если *dir* является относительным путевым именем, то оно определяется относительно *файла-приемника*.

`--owner -o` Сохранение данных о владельце копируемых файлов (только для пользователей с root-правами).

`--perms -p` Сохранение данных о правах доступа к копируемым файлам.

`--recursive -r` Рекурсивный спуск по каталогу, указанному в *файле-источнике*, и копирование всех файлов в иерархии каталогов. См. пример в разделе «Примеры» на с. 605.

`--specials` Копирование специальных файлов.

`--times -t` Сохранение данных о времени последнего изменения копируемых файлов. Этот ключ также ускоряет копирование файлов, поскольку он заставляет rsync не копировать файлы, для которых показатель времени последнего изменения и размер одинаковы как в *файле-источнике*, так и в *файле-приемнике*. См. пример в разделе «Использование замыкающего слэша (/) в файле-источнике» на с. 606.

`--update -u` Пропуск файлов, для которых *файл-приемник* оказался новее *файла-источника*.

`--verbose -v` Вывод информации о том, что в данный момент делает rsync. Этот ключ приносит пользу в сочетании с ключом `--dry-run`. См. пример в разделе «Использование замыкающего слэша (/) в файле-источнике» на с. 606.

`--xattrs -X` Сохранение расширенных атрибутов копируемых файлов. Этот ключ применим не для всех компиляций rsync.

`--compress -z` Сжатие файлов в процессе копирования.

Примечания

У утилиты rsync много ключей. В данной главе рассматриваются лишь некоторые из них. Полный список ключей представлен на man-странице rsync.

OpenSSH По умолчанию rsync копирует файлы на удаленную систему и из нее, используя OpenSSH. Удаленная система должна быть запущена на OpenSSH-сервере. Если есть возможность использования утилиты ssh для регистрации на удаленной системе, то можно воспользоваться rsync для копирования файлов на эту систему или из нее. Если ssh требует ввода пароля, для rsync также потребуется пароль. См. примеры в разделе «Копирование файлов на удаленную систему и из нее» на с. 608. Дополнительные сведения о настройках и использовании OpenSSH даны в части V.

демон rsyncd Если после имени удаленной системы вместо одинарного двоеточия (:) использовать двойное (::), rsync подключится на удаленной системе к демону rsyncd (к утилите, работающей в фоновом режиме, которая не использует OpenSSH). Дополнительные сведения даны на man-странице rsync.

Дополнительная информация

man-страница: rsync

Домашняя страница rsync: www.samba.org/rsync

Вспомогательная информация: www.mikerubel.org/computers/rsync_snapshots

Вспомогательные инструменты: www.rsnapshot.org, backuppc.sourceforge.net

Синхронизация файлов: alliance.seas.upenn.edu/~bcpierce

Примеры

`--recursive` `--verbose` В первом примере rsync создает копию каталога с использованием ключей `--recursive` и `--verbose`. В качестве исходного и целевого выступает рабочий каталог.

```
$ ls -l memos
total 12
-rw-r--r-- 1 max max 1500 Jun  6 14:24 0606
-rw-r--r-- 1 max max 6001 Jun  8 16:16 0608

$ rsync --recursive --verbose memos memos.copy
building file list … done              построение списка … готово
created directory memos.copy             создание каталога memos.copy
memos/
memos/0606
memos/0608
sent 7671 bytes  received 70 bytes  15482.00 bytes/sec
```

```
total size is 7501  speedup is 0.97

$ ls -l memos.copy
total 4
drwxr-xr-x 2 max max 4096 Jul 20 17:42 memos
```

В предыдущем примере rsync копирует каталог memos в каталог memos.copy. Судя по информации, выведенной следующей командой ls, rsync изменила время последнего изменения скопированных файлов на время создания копий:

```
$ ls -l memos.copy/memos
total 12
-rw-r--r-- 1 max max 1500 Jul 20 17:42 0606
-rw-r--r-- 1 max max 6001 Jul 20 17:42 0608
```

Использование замыкающего слэша (/) в файле-источнике

В предыдущем примере показано копирование одного каталога в другой, но могут возникать потребности и в копировании содержимого одного каталога в другой. Замыкающий слэш (/) в *файле-источнике* заставляет rsync работать так, как при указании замыкающих символов /*, и приводит к тому, что rsync копирует содержимое указанного каталога. Замыкающий слэш в *файле-приемнике* ни на что не влияет.

`--times`

В следующем примере с целью сохранить данные о времени последнего изменения копируемых файлов создается еще одна копия каталога memos, но с использованием ключа `--times`. При указании каталога memos используется замыкающий слэш, чтобы в каталог memos.copy2 копировался не сам каталог, а его содержимое.

```
$ rsync --recursive --verbose --times memos/ memos.copy2
building file list … done
created directory memos.copy2
./
0606
0608

sent 7665 bytes  received 70 bytes  15470.00 bytes/sec
total size is 7501  speedup is 0.97
$
$ ls -l memos.copy2
total 12
-rw-r--r-- 1 max max 1500 Jun  6 14:24 0606
-rw-r--r-- 1 max max 6001 Jun  8 16:16 0608
```

`--archive`

Ключ `--archive` заставляет rsync копировать каталоги рекурсивно, разыменовывать символьные ссылки (копировать файлы, на которые они указывают, а не сами символьные ссылки), сохранять данные о времени последнего изменения, о владельце, о связи копируемых фалов с группами и т. д. Этот ключ не заставляет сохранять жесткие ссылки; для этой цели следует использовать ключ `--hard-links`. Дополнительные сведения о ключе `-archive` даны в разделе «Ключи» на с. 603. Следующие команды выполняют те же функции, что и предыдущая:

```
$ rsync --archive --verbose memos/ memos.copy2
$ rsync -av memos/ memos.copy2
```

Удаление файлов

--delete
--dry-run

Ключ --delete заставляет rsync удалить из *файла-приемника* файлы, отсутствующие в *файле-источнике*. Совместное использование ключей --dry-run и --verbose заставляет утилиту показать, чтобы она делала, не будь ключа --dry-run, но при этом rsync не выполняет никаких реальных действий. Использование ключей --dry-run и --verbose вместе с ключом -delete может уберечь от удаления файлов, которые не нужно было бы удалять. Эта комбинация ключей показывает, какие файлы были бы удалены в случае их реального удаления. В следующем примере эти ключи используются в дополнение к ключу --archive.

```
$ ls -l memos memos.copy3
memos:
total 20
-rw-r--r-- 1 max max 1500 Jun  6 14:24 0606
-rw-r--r-- 1 max max 6001 Jun  8 16:16 0608
-rw-r--r-- 1 max max 5911 Jun 10 12:02 0610

memos.copy3:
total 16
-rw-r--r-- 1 max max 1500 Jun  6 14:24 0606
-rw-r--r-- 1 max max 6001 Jun  8 16:16 0608
-rw-r--r-- 1 max max 1200 Jul 21 10:17 notes

$ rsync --archive --verbose --delete --dry-run memos/ memos.copy3
building file list … done
deleting notes                  Удаление файла notes
./
0610
sent 125 bytes  received 32 bytes  314.00 bytes/sec
total size is 13412  speedup is 85.43
```

Утилита rsync показывает, что если ее запустить без ключа --dry-run, будет удален файл notes. Она также сообщает, что будет скопирован файл 0610.

ЧТОБЫ ПОНЯТЬ, КАКОВ БУДЕТ РЕЗУЛЬТАТ, ЛУЧШЕ ПРОВЕСТИ ПРОВЕРКУ — УДАЛЕНИЕ МОЖЕТ ПРИВЕСТИ К НЕОЖИДАННЫМ ПОСЛЕДСТВИЯМ

Если в файле-источнике будет пропущен нужный слэш (/) или поставлен ненужный, использование ключа --delete может запросто привести к удалению целого дерева каталогов. Чтобы протестировать работу команды rsync, используйте ключ --delete с ключами --dry-run и --verbose.

Если использование длинных версий ключей надоест, можно воспользоваться их краткими однобуквенными версиями. Следующая команда rsync является аналогом предыдущей (а короткой версии ключа --delete просто не существует):

```
$ rsync -avn --delete memos/ memos.copy3
```

В следующем примере запускается такая же команда rsync, но без ключа --dry-run. Команда ls показывает результаты работы команды rsync: ключ --delete заставляет rsync удалить файл notes из *файла-приемника* (memos.copy3), поскольку его нет в *файле-источнике* (memos). Кроме этого, rsync копирует файл 0610.

```
$ rsync --archive --verbose --delete memos/ memos.copy3
building file list … done
```

```
deleting notes
./
0610

sent 6076 bytes  received 48 bytes  12248.00 bytes/sec
total size is 13412  speedup is 2.19

$ ls -l memos memos.copy3
memos:
total 20
-rw-r--r-- 1 max max 1500 Jun  6 14:24 0606
-rw-r--r-- 1 max max 6001 Jun  8 16:16 0608
-rw-r--r-- 1 max max 5911 Jun 10 12:02 0610

memos.copy3:
total 20
-rw-r--r-- 1 max max 1500 Jun  6 14:24 0606
-rw-r--r-- 1 max max 6001 Jun  8 16:16 0608
-rw-r--r-- 1 max max 5911 Jun 10 12:02 0610
```

До сих пор во всех примерах файлы копировались локально. Для копирования файлов в другие каталоги нужно заменить простые имена файлов относительными или абсолютными именами. На локальной системе относительные имена определяются относительно рабочего каталога, а на удаленной — относительно личного каталога пользователя. Например, следующая команда копирует memos из рабочего каталога в каталог /backup на локальной системе:

```
$ rsync --archive --verbose --delete memos/ /backup
```

Копирование файлов на удаленную систему и из нее

Для копирования файлов на удаленную систему и из нее эта система должна быть запущена на OpenSSH-сервере или на каком-нибудь другом механизме пересылки, к которому может подключиться rsync. Дополнительные сведения указаны в разделе «Примечания» на с. 605. Для указания файла на удаленной системе перед именем файла нужно поставить имя удаленной системы и двоеточие. Относительные имена на удаленной системе указываются относительно личного каталога пользователя. Абсолютные имена ведут свое начало от корневого каталога. Дополнительные сведения об относительных и абсолютных именах даны в главе 4.

В следующем примере пользователь Max копирует каталог memos из рабочего каталога на локальной системе в каталог holdfiles на удаленной системе по имени coffee. Утилита ssh запускает на coffee команду ls, чтобы показать результат работы команды rsync. Утилиты rsync и ssh не требуют ввода пароля, потому что Max настраивает основанные на OpenSSH утилиты на автоматический вход в систему coffee (см. часть V, раздел «ssh», подраздел «Ключи авторизации: автоматический вход в систему» на с. 834).

```
$ rsync --archive memos/ coffee:holdfiles
$ ssh coffee 'ls -l holdfiles'
total 20
-rw-r--r-- 1 max max 1500 Jun  6 14:24 0606
-rw-r--r-- 1 max max 6001 Jun  8 16:16 0608
-rw-r--r-- 1 max max 5911 Jun 10 12:02 0610
```

При копировании с удаленной системы на локальную нужно поставить имя удаленной системы перед *файлом-источником*:

```
$ rsync --archive coffee:holdfiles/ ~/memo.copy4
$ rsync --archive coffee:holdfiles/ /home/max/memo.copy5
```

Обе эти команды копируют каталог holdfiles из рабочего каталога пользователя Max на системе coffee в его личный каталог на локальной системе. При работе под управлением Mac OS X нужно вместо /home поставить /Users.

Зеркальное копирование каталога

Утилиту rsync можно использовать для обслуживания копии каталога. Поскольку это будет абсолютно точная копия, ее еще называют *зеркалом*. Зеркальный каталог должен находиться на OpenSSH-сервере (к нему должна быть возможность подключиться с помощью OpenSSH-утилиты ssh). Если нужно запустить этот сценарий с помощью crontab (см. часть V, раздел «crontab» на с. 662), следует настроить OpenSSH на автоматический вход в удаленную систему (без предоставления пароля; см. часть V, раздел «ssh», подраздел «Ключи авторизации: автоматический вход в систему» а с. 834).

--compress
--update

Следующий пример показывает использование rsync-ключей --compress и --update. Ключ --compress заставляет rsync сжимать файлы в процессе их копирования, что зачастую приводит к более быстрой передаче данных. В некоторых случаях, когда, к примеру, имеется установка с быстродействующей сетью и медленным центральным процессором, сжатие файлов может замедлить передачу. Ключ --update не дает rsync переписывать более новые файлы более старыми.

Как и в случае использования любых других сценариев оболочки, для работы со сценарием mirror нужны права на чтение и выполнение того файла, в котором он хранится. Чтобы упростить чтение, каждый ключ в сценарии располагается в отдельной строке. Каждая строка каждой команды, за исключением последней, завершается пробелом и обратным слэшем (\). Пробел отделяет ключ от следующего ключа, а обратный слэш отключает действие следующего разделителя строк, поэтому оболочка передает все аргументы утилите rsync и не интерпретирует разделители строк как конец команды.

```
$ cat mirror
rsync \
--archive \
--verbose \
--compress \
--update \
--delete \
~/mirrordir/ coffee:mirrordir

$ ./mirror > mirror.out
```

В этом примере вывод команды mirror перенаправляется для просмотра в файл mirror.out. Если нужно, чтобы команда не выводила ничего, кроме сообщений об ошибках, ключ --verbose следует убрать. Команда rsync в сценарии mirror копирует иерархию каталога mirrordir из личного каталога пользователя на локальной системе в личный каталог пользователя на удаленной системе (на сервере). В данном примере удаленная система называется coffee. Присутствие ключа --update заставляет rsync не

переписывать самые последние версии файлов на сервере более старыми версиями таких же файлов, взятых из локальной системы. Хотя при постоянно неизменном состоянии файлов на сервере этот ключ и не нужен, он может уберечь от неприятностей, если на серверной системе файлы будут случайно обновлены или добавлены. Ключ `--delete` заставляет rsync удалить на серверной системе файлы, отсутствующие на локальной системе.

Создание резервных копий

После проведения исходного полноценного резервного копирования rsync может эффективно осуществлять дополняющее резервное копирование, экономя при этом рабочее время машины и пространство накопителя. При дополняющем (инкрементном) копировании сохраняются только те файлы, которые претерпели изменения со времени последнего резервного копирования; именно их и нужно копировать утилите rsync. Как показано в следующем примере, rsync без использования дополнительного дискового пространства может превращать каждое дополняющее резервное копирование в полноценное, путем создания жестких ссылок между дополняющими резервными копиями и неизменившимися файлами, имеющимися в исходной полноценной резервной копии.

`--link-dest=`*dir* С помощью ключа `--link-dest=`*dir* утилита rsync превращает резервное копирование в простую и эффективную операцию. Она предоставляет пользователю и (или) системному администратору возможность делать текущие копии, которые превращаются в полноценные при использовании минимально необходимого дополнительного пространства носителя вдобавок к уже занятому исходной резервной копией. Каталог *dir* всегда находится на машине, хранящей *файл-приемник*. Если *dir* является относительным именем, то оно определяется относительно *файла-приемника*. Описание этого ключа дано в разделе «Ключи» на с. 603.

Следующий код является простой командой rsync, в которой используется ключ `--link-dest=`*dir*:

```
$ rsync --archive --link-dest=../backup source/ destination
```

При запуске этой команды rsync проходит сверху вниз по содержимому каталога source, проверяя каждый найденный файл. Для каждого файла в каталоге source rsync ищет в каталоге destination точную копию этого файла.

- Если она находит точную копию файла в каталоге destination, утилита rsync переходит к следующему файлу.
- Если она не находит точную копию файла в каталоге destination, утилита rsync ищет точную копию этого файла в каталоге backup.
  - Если она находит точную копию файла в каталоге backup, утилита rsync создает жесткую ссылку из файла в каталоге backup на файл в каталоге destination.
  - Если она не находит точную копию файла в каталоге backup, утилита rsync копирует файл из каталога source в каталог destination.

Следующий код является простым примером, показывающим, как используется rsync с ключом `--link-dest=`*dir* для создания полноценной и дополняющей резервных

копий. Хотя файлы резервной копии находятся на локальной системе, их легко можно разместить и на удаленной системе.

Согласно двум аргументам rsync, используемым в сценарии bkup, rsync копирует каталог memos в каталог bu.0. Ключ `--link-dest=`*dir* заставляет rsync проверять существование каждого файла, подлежащего копированию, в каталоге bu.1. Если такой файл существует, rsync не копирует его, а создает ссылку на файл, существующий в каталоге bu.1.

Сценарий bkup попеременно работает с тремя каталогами резервного копирования с именами bu.0, bu.1 и bu.2 и вызывает rsync. Сценарий удаляет bu.2, перемещает bu.1 в bu.2, а затем перемещает bu.0 в bu.1. При первом запуске сценария rsync копирует все файлы из каталога memos, потому что их нет в каталоге bu.0 или в каталоге bu.1.

```
$ cat bkup
rm -rf bu.2
mv bu.1 bu.2
mv bu.0 bu.1
rsync --archive --link-dest=../bu.1 memos/  bu.0
```

Перед первым запуском bkup каталогов bu.0, bu.1 и bu.2 не существует. Благодаря использованию ключа `-f` rm не выводит сообщение об ошибке при попытке переместить несуществующий каталог bu.2. Пока bkup создает bu.0 и bu.1, mv выводит сообщения об ошибках `No such file or directory` (Файл или каталог с таким именем отсутствует).

В следующем примере ls показывает сценарий bkup и содержимое каталога memos. После запуска bkup команда ls показывает содержимое memos и нового каталога bu.0, в котором содержатся точно такие же файлы, что и в каталоге memos. Утилита rsync не создает ссылок из-за отсутствия файлов в каталоге bu.1, поскольку такого каталога не существует.

```
$ ls -l *
-rwxr-xr-x 1 max max   87 Jul 22 11:23 bkup

memos:
total 20
-rw-r--r-- 1 max max 1500 Jun  6 14:24 0606
-rw-r--r-- 1 max max 6001 Jun  8 16:16 0608
-rw-r--r-- 1 max max 5911 Jun 10 12:02 0610

$ ./bkup
mv: cannot stat 'bu.1': No such file or directory
mv: cannot stat 'bu.0': No such file or directory

$ ls -l *
-rwxr-xr-x 1 max max   87 Jul 22 11:23 bkup

bu.0:
total 20
-rw-r--r-- 1 max max 1500 Jun  6 14:24 0606
-rw-r--r-- 1 max max 6001 Jun  8 16:16 0608
-rw-r--r-- 1 max max 5911 Jun 10 12:02 0610
```

```
memos:
total 20
-rw-r--r-- 1 max max 1500 Jun  6 14:24 0606
-rw-r--r-- 1 max max 6001 Jun  8 16:16 0608
-rw-r--r-- 1 max max 5911 Jun 10 12:02 0610
```

После работы с файлами в каталоге memos команда ls показывает, что файл 0610 был удален, а файл newfile — добавлен:

```
$ ls -l memos
total 20
-rw-r--r-- 1 max max 2100 Jul 22 14:31 0606
-rw-r--r-- 1 max max 6001 Jun  8 16:16 0608
-rw-r--r-- 1 max max 5251 Jul 22 14:32 newfile
```

После нового запуска bkup каталог bu.0 хранит те же файлы, которые находятся в каталоге memos, а каталог bu.1 хранит те файлы, которые хранились в bu.0 до запуска bkup. Файл 0608 не изменялся, поэтому rsync с ключом `--link-dest=`*dir* его не копирует, а делает ссылку из копии в bu.1 на копию в bu.0, что показано цифрой 2, которую команда ls выводит между полем прав доступа и max.

```
$ ./bkup
mv: cannot stat 'bu.1': No such file or directory

$ ls -l bu.0 bu.1
bu.0:
total 20
-rw-r--r-- 1 max max 2100 Jul 22 14:31 0606
-rw-r--r-- 2 max max 6001 Jun  8 16:16 0608
-rw-r--r-- 1 max max 5251 Jul 22 14:32 newfile

bu.1:
total 20
-rw-r--r-- 1 max max 1500 Jun  6 14:24 0606
-rw-r--r-- 2 max max 6001 Jun  8 16:16 0608
-rw-r--r-- 1 max max 5911 Jun 10 12:02 0610
```

Достоинство такой организации заключается в том, что каждое дополняющее резервное копирование занимает только то дисковое пространство, которое нужно для содержания измененных файлов. Файлы, не подвергавшиеся изменениям, хранятся в виде ссылок, занимающих минимальное дисковое пространство. При этом пользователи и системный администратор имеют доступ к каталогу, в котором содержится полноценная резервная копия.

Такой сценарий резервного копирования, как bkup, можно запускать каждый час, каждый день или с любой другой нужной периодичностью. Если rsync не требует введения пароля, этот процесс можно автоматизировать, воспользовавшись утилитой crontab (см. часть V).

Заключение

Утилита rsync копирует обычный файл или иерархию каталогов на локальной системе или между локальной и удаленной системами по сети. По умолчанию эта утилита использует для передачи файлов openSSH и такой же механизм аутентификации, как и openSSH; поэтому она предоставляет такую же степень безопасности, что и openSSH. Если утилите rsync нужен пароль, она выводит приглашение на его ввод.

Упражнения

1. Назовите три особенности rsync.
2. Напишите команду rsync, которая копирует каталог backmeup из вашего личного каталога на локальной системе в каталог /tmp на системе coffee, сохраняя данные о владельце файлов, о правах доступа к файлам и о времени их последнего изменения. Напишите команду, копирующую тот же каталог в ваш личный каталог на системе coffee, предполагая, что ваш личный каталог на локальной системе не является рабочим каталогом.
3. Вы пишите команду rsync, в которой используется ключ `--delete`. Какие ключи нужно использовать для тестирования команды без реального копирования или удаления файлов?
4. Для чего предназначен ключ `--archive`? В чем заключается польза от его применения?)
5. Как при запуске сценария, аналогичного bkup (см. раздел «Создание резервных копий» на с. 610), для резервного копирования файлов на удаленную систему можно осуществлять перемещение (переименование) файлов на удаленной системе?
6. К какому эффекту приводит использование замыкающего слэша (/) в *файле-источнике*?

ЧАСТЬ V

Справочник команд

Указатель команд

В следующих таблицах перечислены утилиты, рассмотренные в этой части книги, сгруппированные по функциям и внутри функций по алфавиту. Хотя в большинстве своем это настоящие утилиты (программы, отделенные от оболочки), некоторые из них встроены в оболочки (то есть являются встроенными командами). Утилита sample показывает формат с описанием каждой утилиты, рассматриваемой в данной части книги.

Утилиты для работы с файлами и для их вывода на экран

aspell	Проверка файлов на правописание — с. 622
bzip2	Сжатие и восстановление файлов — с. 630
cat	Объединение и вывод содержимого файлов — с. 633
cmp	Сравнение двух файлов — с. 647
comm	Сравнение отсортированных файлов — с. 649
cp	Копирование файлов — с. 653
cpio	Создание архива, восстановление файлов из архива или копирование иерархии каталога — с. 657
cut	Выбор символов или полей из строк ввода — с. 665
dd	Преобразование и копирование файла — с. 669
diff	Вывод разницы между двум текстовыми файлами — с. 674
ditto	Копирование файлов, создание и распаковка архивов — с. 683
emacs	Редактор — с. 220
find	Поиск файлов на основе определенного критерия — с. 699
fmt	Очень простое форматирование текста — с. 708
grep	Поиск шаблона в файлах — с. 729
gzip	Сжатие и восстановление файлов — с. 734
head	Вывод начала файла — с. 737
less	Поэкранный вывод текстовых файлов — с. 744
ln	Создание ссылки на файл — с. 748
lpr	Отправка файлов на принтеры — с. 750
ls	Вывод информации об одном или нескольких файлах — с. 753
man	Вывод документации по командам — с. 767
mkdir	Создание каталога — с. 771
mv	Переименование или перемещение файла — с. 778
od	Вывод на экран содержимого файла — с. 782
open	Открытие файлов, каталогов и URL-адресов — с. 786
otool	Вывод файлов объектов, библиотек и исполняемых файлов — с. 788
paste	Объединение соответствующих строк из файлов — с. 790
pax	Создание архива, восстановление файлов из архива или копирование иерархии каталога — с. 792
plutil	Работа с файлами списка свойств — с. 798

pr	Разбиение файлов на страницы для печати — с. 799
rm	Удаление файлов (удаление ссылок) — с. 810
rmdir	Удаление каталогов — с. 811
sed	Неинтерактивное редактирование файла — с. 816
sort	Сортировка и (или) объединение файлов — с. 820
split	Разбиение файла на разделы — с. 829
strings	Вывод строк печатаемых символов — с. 839
tail	Вывод последней части (хвоста) файла — с. 846
touch	Создание файла или изменение прав доступа к нему и (или) времени последнего изменения — с. 859
uniq	Вывод уникальных строк — с. 868
vim	Редактор — с. 168
wc	Вывод номеров строк, слов и байтов — с. 872

Сетевые утилиты

ftp	Передача файлов по сети — с. 715
rcp	Копирование одного или нескольких файлов на удаленную систему или из нее — с. 805
rlogin	Вход на удаленную систему — с. 809
rsh	Выполнение команд на удаленной системе — с. 812
rsync	Безопасное копирование файлов и иерархии каталогов по сети — с. 814
scp	Безопасное копирование одного или нескольких файлов на удаленную систему или из нее — с. 814
ssh	Безопасное выполнение команд на удаленной системе — с. 830
telnet	Подключение к удаленной системе по сети — с. 850

Утилиты, которые выводят и изменяют статус

cd	Смена рабочего каталога — с. 635
chgrp	Изменение группы, связанной с файлом — с. 636
chmod	Изменение режима доступа (прав) к файлу — с. 640
chown	Изменение владельца файла и (или) группы, с которой он связан — с. 645
date	Вывод или установка системного времени и даты — с. 667
df	Вывод сведений об использовании дискового пространства — с. 672
dscl	Вывод сведений о Directory Service и работа с ними — с. 686 (OS X)
dmesg	Вывод сообщений ядра — с. 685
du	Вывод информации об использовании диска иерархией каталогов и (или) файлами — с. 689
file	Вывод классификации файла — с. 697
finger	Вывод сведений о пользовтелях — с. 706
Get-FileInfo	Вывод атрибутов файла — с. 727

kill	Завершение процесса с помощью PID — с. 738
killall	Завершение процесса с указанием его имени — с. 740
nice	Изменение приоритета команды — с. 780
nohup	Запуск команды, которая продолжает свое выполнение после вашего выхода из системы — с. 782
ps	Отображение статуса процесса — с. 801
renice	Изменение приоритета процесса — с. 807
SetFile	Установка атрибутов файла — с. 817
sleep	Создание процесса, приостанавливающего свою работу на указанный интервал времени — с. 818
stat	Вывод сведений о файлах — с. 838
stty	Вывод или установка параметров терминала — с. 840
sysctl	Вывод и изменение значений переменных ядра — с. 845
top	Динамическое отображение статуса процесса — с. 852
umask	Установка маски прав доступа, определяемой при создании файла — с. 867
w	Вывод сведений о пользователях системы — с. 870
which	Отображение местонахождения команды в PATH — с. 873
who	Вывод сведений о вошедших в систему пользователях — с. 875

Утилиты, являющиеся средствами программирования

awk	Поиск соответствия шаблонам и обработка в файле соответствующего им содержимого — с. 552
configure	Автоматическое конфигурирование исходного кода — с. 651
gawk	Поиск соответствия шаблонам и обработка в файле соответствующего им содержимого — с. 722
gcc	Компилирование программ на C и C++ — с. 723
make	Поддержка актуальности набора программ — с. 761
mawk	Поиск соответствия шаблонам и обработка в файле соответствующего им содержимого — с. 552

Утилиты разного назначения

at	Выполнение команд в указанное время — с. 626
cal	Вывод календаря — с. 632
crontab	Обслуживание файлов crontab — с. 662
diskutil	Проверка, изменение и восстановление локальных томов — с. 680
echo	Вывод сообщения — с. 691
expr	Вычисление выражения — с. 694
fsck	Проверка и восстановление файловой системы — с. 710
launchctl	Управление демоном launchd — с. 741
mkfs	Создание на устройстве файловой системы — с. 772
Mtools	Использование команд DOS-стиля для работы с файлами и каталогами — с. 775

tee	Копирование стандартного ввода на стандартный вывод и одного или нескольких файлов — с. 849
test	Вычисление выражения — с. 852
tr	Замена указанных символов — с. 861
tty	Вывод путевого имени терминала — с. 864
tune2fs	Изменение параметров на файловой системе ext2, ext3 или ext4 — с. 864
xargs	Превращение стандартного ввода в командные строки — с. 877

Стандартные мультипликативные суффиксы

Ряд утилит допускает после указания количества байт использовать суффиксы, перечисленные в табл. V.1. Перед мультипликативными суффиксами можно поставить умножаемое ими число. Например, 5K означает 5×2^{10}. Отсутствие множителя свидетельствует о том, что мультипликативный суффикс нужно умножать на 1. Утилиты, использующие эти суффиксы, отмечаются особым образом.

Таблица V.1. Мультипликативные суффиксы

Суффикс	Множитель	Суффикс	Множитель
KB	1000 (10^3)	PB	10^{15}
K	1024(2^{10})	P	2^{50}
MB	1 000 000 (10^6)	EB	10^{18}
M	1 048 576 (2^{20})	E	2^{60}
GB	1 000 000 000 (10^9)	ZB	10^{21}
G	1 073 741 824 (2^{30})	Z	2^{70}
TB	10^{12}	YB	10^{24}
T	2^{40}	Y	2^{80}

BLOCKSIZE

При работе в Mac OS X, некоторые утилиты используют переменную среды окружения BLOCKSIZE для установки размера блока по умолчанию. Переменной BLOCKSIZE можно присвоить значение, являющееся количеством байт, или значение, использующее один из суффиксов K, M или G. Утилиты, использующие BLOCKSIZE, способны идентифицировать этот текст.

Общие ключи

Некоторые GNU-утилиты используют общие ключи, перечисленные в табл. V.2. Утилиты, использующие эти ключи, отмечаются особым образом.

Таблица V.2. Общие ключи командной строки

Ключ	Действие
-	Одинарный дефис, появляющийся вместо имени файла, заставляет утилиту принимать данные не из файла, а со стандартного ввода
--	Двойной дефис является признаком конца ключей в командной строке. После этого ключа можно ставить аргументы, начинающиеся с дефиса. Без этого ключа утилита предполагает, что аргумент, начинающийся с дефиса, является ключом

продолжение

Таблица V.2 (*продолжение*)

Ключ	Действие
`--help`	Вывод справочного сообщения по утилите. Среди таких сообщений попадаются и очень длинные. Используя канал, можно пропустить вывод через утилиту less, чтобы выводить его в поэкранном режиме. Можно, например, дать команду `ls --help | less`. Кроме этого, если нужно найти вполне определенную информацию, можно, используя канал, пропустить вывод через утилиту grep. Например, можно дать следующую команду для получения информации о ключе `-d` утилиты ls: `ls --help | grep -- -d`. Сведение о двойном дефисе даны в предыдущей записи этой таблицы
`--version`	Вывод сведений о версии утилиты

Утилита-пример

Следующее описание утилиты sample (пример) показывает формат этой части книги, который используется для описания утилит. Эти описания похожи на те, что даются на man-страницах (см. раздел «man: Отображение системного руководства» на с. 53 и описание утилиты man в данной части); но большинство пользователей считают, что описания в данной книге легче читаются и изложены более понятным языком. В них подчеркнуты наиболее полезные свойства утилит и зачастую опущены их ничем особо не примечательные и менее востребованные свойства. Сведения о редко востребуемых свойствах можно найти на man- и info-страницах или в сообщении, выводимом при вызове утилиты с ключом `--help`, который поддерживается многими утилитами.

sample

Метка (OS X), свидетельствует о том, что эта утилита работает только под управлением Mac OS X.

Краткое описание предназначения утилиты:

*sample [ключи] **аргументы***

За строкой синтаксиса следует описание утилиты. Строка синтаксиса показывает, как нужно запускать утилиту из командной строки. *Ключи* и *аргументы*, заключенные в квадратные скобки (`[]`), являются необязательными. Слова, которые указаны в синтаксисе *курсивом*, вводятся именно в таком виде, в котором они указаны. Слова, вместо которых при наборе команды нужно поставить конкретные значения, указаны в синтаксисе ***полужирным курсивом***. Слова, перечисляемые в качестве аргументов команды, идентифицируют одиночные аргументы (например, ***файл-источник***) или группу однотипных аргументов (например, ***список-каталогов***).

Аргументы

В этом разделе дается описание аргументов, которые могут использоваться при запуске утилиты. Сам аргумент, как показано в предыдущей строке синтаксиса, набран ***полужирным курсивом***.

Ключи

В этом разделе дается описание ключей, которые могут быть использованы в команде. Если не указано ничего иного, перед ключами должен ставиться один или два дефиса. Многие команды воспринимают один дефис, стоящий сразу перед несколькими ключами (см. раздел «Ключи» на с. 138). Ключи в данном разделе следуют в алфавитном порядке, определяемом короткими (использующими одинарный дефис) ключами. Если у ключа имеется только длинная версия (с двойным дефисом), то порядок выстраивается по длинным версиям ключа. Примеры некоторых ключей выглядят следующим образом:

`--delimiter=`*dchar*
`-d` *dchar*
: Этот ключ включает в себя аргумент. Аргумент набран ***полужирным курсивом*** как в заголовке, так и в описании. Любые аргументы, набранные этим шрифтом, должны быть заменены каким-нибудь другим словом (именем файла, строкой символов или другим значением). Символы, набранные моноширинным шрифтом (например, `--delimiter` и `-d`), вставляются в командную строку без изменений.

`--make-dirs`
`-m`
: у этого ключа имеется длинная и короткая версии. Можно использовать любую из них, поскольку они абсолютно эквивалентны. Это описание ключа заканчивается меткой (Linux), значит, данный ключ доступен только при работе с Linux. Ключи, за описаниями которых не следуют метки Linux или OS X, доступны при работе с любой из этих операционных систем. (Linux)

`-t`
: (содержание). Это простой ключ, перед которым стоит одинарный дефис и за которым не следуют аргументы. У него нет длинной версии. Содержание, показываемое в круглых скобках в начале описания, — это ключевой признак, намекающий на предназначение буквы этого ключа. Это описание ключа заканчивается меткой (OS X), показывающей, что ключ доступен только при работе с OS X. Ключи, за описаниями которых не следуют метки Linux или OS X, доступны при работе с любой из этих операционных систем. (OS X)

Комментарий

Этот необязательный раздел описывает порядок использования утилиты и любые, имеющиеся у нее интересные особенности.

Примечания

Этот раздел содержит примечания общего плана, как важные, так и не очень.

Примеры

Этот раздел содержит примеры, показывающие как нужно использовать утилиту. Он имеет обучающий характер, поэтому в нем используется более свободный стиль изложения по сравнению с предыдущими разделами описания.

aspell

Проверка файла на ошибки правописания.

aspell check [ключи] имя_файла
aspell list [ключи] < имя_файла
aspell config
aspell help

Утилита aspell проверяет правописание слов в документе по стандартному словарю. Эту утилиту можно использовать в интерактивном режиме: она выводит каждое неправильно написанное слово в контексте наряду с меню, дающим возможность принять слово в его собственном виде, выбрать для замены один из предлагаемых aspell вариантов написания слова, вставить слово в свой персональный словарь или заменить слово тем, которое будет введено вручную. Утилиту aspell можно также использовать в пакетном режиме, чтобы она читала данные со стандартного ввода и записывала их в стандартный вывод. Эта утилита доступна только при работе с Linux. (Linux)

ИЗ-ЗА СВОЕГО ВВОДА ASPELL НЕ ПОХОЖА НА ДРУГИЕ УТИЛИТЫ

в отличие от многих других утилит, aspell не воспринимает ввод из стандартного ввода, когда в командной строке не указано имя_файла. Источник ввода для aspell определяется указанием характера ее действия.

Действия

При запуске aspell нужно выбрать только одно *действие*.

`check -c` Запуск aspell в качестве интерактивного средства проверки правописания. Вход поступает только из одного файла, имя которого указано в командной строке. См. далее раздел «Комментарий».

`config` Вывод конфигурации aspell с демонстрацией как значений по умолчанию, так и текущих значений. Отправьте вывод через канал утилите less для упрощения просмотра или воспользуйтесь утилитой grep для поиска нужного ключа (например, `aspell config | grep backup`).

`help -?` Вывод подробной справочной страницы. Для упрощения просмотра отправьте вывод через канал утилите less.

`list -l` Запуск aspell в пакетном (неинтерактивном) режиме со вводом, поступающим со стандартного ввода, и с выводом, отправляемым на стандартный вывод.

Аргументы

Имя_файла — это имя того файла, содержимое которого нужно проверить. Утилита aspell допускает использование этого аргумента только с действием `check` (`-c`). Если используется действие `list` (`-l`), ввод должен поступать со стандартного ввода.

Ключи

С утилитой aspell используется большое количество ключей. В данном разделе перечисляются лишь некоторые, наиболее востребованные ключи, а полный список

приведен на man-странице, посвященной aspell. Значения по умолчанию для многих ключей определяются при компиляции aspell (см. действие config).

Ключи можно указать в командной строке, а также в качестве значения переменной оболочки ASPELL_CONF или в своем личном конфигурационном файле (~/.aspell.conf). Пользователь, работающий с root-правами, может создать глобальный конфигурационный файл (/etc/aspell.conf). В конфигурационном файле каждый ключ должен находиться на отдельной строке, а в переменной ASPELL_CONF ключи должны отделяться друг от друга точкой с запятой (;). Ключи в командной строке отменяют соответствующие ключи, указанные в переменной ASPELL_CONF, а те, в свою очередь, отменяют соответствующие ключи в вашем личном конфигурационном файле, которые отменяют соответствующие им ключи в глобальном конфигурационном файле.

Следующий список состоит из двух типов ключей: имеющих булевы и обычные значения. Булевы ключи включают свойство (разрешают его использование) или выключают его (запрещают его использование). Чтобы выключить свойство, нужно поставить перед булевым ключом приставку dont-. Например, ключ --ignore-case включает свойство игнорирования регистра букв, а ключ --dont-ignore-case выключает это свойство.

Ключи, имеющие обычные значения, присваивают свойству какое-нибудь значение. К ним относятся ключи, имеющие знак равенства и значение, например --ignore=4.

Для всех ключей в конфигурационном файле или в переменной ASPELL_CONF дефисы следует опускать (ignore-case или dont-ignore-case).

КЛЮЧИ ASPELL И ЛИДИРУЮЩИЕ ДЕФИСЫ

Ключи указываются по-разному, в зависимости от места их указания: в командной строке, в переменной оболочки ASPELL_CONF или в конфигурационном файле. В командной строке перед длинными ключами ставится двойной дефис (например, --ignore-case или --dont-ignore-case). В ASPELL_CONF и в конфигурационных файлах лидирующие дефисы отбрасываются (например, ignore-case или dont-ignore-case).

--dont-backup Отказ от создания резервного файла с именем *имя_файла*.bak (по умолчанию, если выбрано действие check, применяется ключ --backup).

--ignore=*n* Игнорирование слова с *n* или меньшим количеством символов (по умолчанию используется значение 1).

--ignore-case Игнорирование регистра букв в проверяемых словах (по умолчанию используется ключ --dont-ignore-case).

--lang=*cc* Указание двухбуквенного кода языка (***cc***). По умолчанию для кода языка используется значение переменной LC_MESSAGES (см. главу 8, табл. 8.5).

--mode=*mod* Указание используемого фильтра. Значение ***mod*** выбирается из url (по умолчанию), none, sgml и др. Режимы работают следующим образом: url: пропуск URL-адресов, имен хостов и адресов электронной почты; none: выключение всех фильтров; sgml: пропуск команд SGML, HTML, XHTML и XML.

--strip-accents Удаление знаков ударений из всех слов в словаре перед проверкой слов (по умолчанию используется ключ --dont-strip-accents).

Комментарий

Утилита aspell имеет два основных режима работы: пакетный и интерактивный. Пакетный режим указывается за счет использования действия `list` или `-l`. В пакетном режиме aspell допускает использование документов, которые нужно проверить на ошибки правописания в качестве стандартного ввода, и отправляет список потенциально неправильно написанных слов на стандартный вывод.

Интерактивный режим указывается за счет использования действия `check` или `-c`. В интерактивном режиме aspell выводит на экран потенциально неправильно написанные слова, выделенные в контексте, и меню выбора. Ее работа проиллюстрирована в разделе «Примеры» на с. 625. В меню включены различные команды (см. табл. V.3), а также некоторые предложения похожих, правильно написанных слов. Нужно либо ввести одно из чисел меню для выбора предлагаемого слова для замены слова, вызывающего вопросы, либо ввести букву, чтобы дать команду.

Таблица V.3. Команды aspell

Команда	Действие
ПРОБЕЛ	Отказ от действий и переход к следующему неправильно написанному слову
N	Замена неправильно написанного слова предлагаемым словом под номером *n*
A	Добавление «неправильно» написанного слова в персональный словарь
B	Преждевременный выход из aspell без сохранения изменений
i или I (буква «i»)	Игнорирование неправильно написанного слова. I (буква «I» в верхнем регистре) приводит к игнорированию всех появлений этого слова; а i — к игнорированию только данного его появления, и по действию аналогична нажатию пробела
l (буква «l» в нижнем регистре)	Перевод букв «неправильно» написанного слова в нижний регистр и добавление его в персональный словарь
r или R	Замена неправильно написанного слова тем словом, которое введено вами в нижней части экрана. R приводит к замене всех появлений этого слова; а r — к замене только данного его появления
x	Сохранение файла откорректированным по состоянию на данный момент, и выход из aspell

Примечания

Дополнительную информацию можно получить на главной странице aspell на вебсайте aspell.net и в каталоге /usr/share/doc/aspell.

Утилита aspell не является абсолютно надежным и легким способом обнаружения ошибок правописания. К тому же она не выявляет неверно использованных, но при этом правильно написанных слов (например, слова `red` вместо слова `read`).

Проверка правописания из emacs

Использование aspell из emacs можно существенно упростить, добавив к своему файлу ~/.emacs (см. раздел «Запуск .emacs» на с. 223) следующую строку. Эта строка заставляет имеющуюся в emacs функцию ispell вызывать aspell:

```
(setq-default ispell-program-name "aspell")
```

Проверка правописания из vim

Можно упростить использование aspell и из vim, добавив следующую строку к своему файлу ~/.vimrc (см. раздел «Файл запуска .vimrc» на с. 204):

```
map ^T :w!<CR>:!aspell check %<CR>:e! %<CR>
```

При вводе этой строки в ~/.vimrc с использованием редактора vim ^T следует вводить с помощью CONTROL+V CONTROL+T (см. раздел «Цитирование специальных символов в режиме ввода» на с. 189). При наличии этой строки в файле ~/.vimrc CONTROL+T приводит к привлечению aspell для проверки правописания редактируемого в vim файла.

Примеры

В следующих примерах aspell используется для исправления правописания в файле memo.txt:

```
$ cat memo.txt
Here's a document for teh aspell utilitey
to check. It obviosly needs proofing
quiet badly.
```

В первом примере aspell используется в check и без ключей. В нем показан внешний вид экрана для первого неправильно написанного слова teh. В нижней части экрана представлено меню команд и предлагаемые слова. Каждое из пронумерованных слов немного отличается от неправильно написанного слова:

```
$ aspell check memo.txt

Here's a document for teh aspell utilitey
to check. It obviosly needs proofing
quiet badly.

------------------------------------------------------
1) the                                  6) th
2) Te                                   7) tea
3) tech                                 8) tee
4) Th                                   9) Ted
5) eh                                   0) tel
i) Ignore                               I) Ignore all
r) Replace                              R) Replace all
a) Add                                  l) Add Lower
b) Abort                                x) Exit
------------------------------------------------------
?
```

В ответ на предыдущее отображение нужно ввести один из пунктов меню; aspell выполнит ваше распоряжение и переместит выделение на следующее неправильно написанное слово (если только не будет выбран пункт преждевременного прекращения работы или выхода). В данном случае ввод цифры 1 (один) изменит в файле teh на the.

В следующем примере используется действие `list` для вывода списка неправильно написанных слов. Слово `quiet` в этом списке отсутствует: оно применено неправильно, но не имеет ошибок правописания.

```
$ aspell list < memo.txt
teh
aspell
utilitey
obviosly
```

В последнем примере также используется действие `list`. В нем показан быстрый способ проверки правописания слова или двух слов с помощью одной команды. Пользователь дает aspell команду `list`, а затем вводит `seperate temperature` в стандартный ввод aspell (то есть осуществляет ввод с клавиатуры). После того как пользователь нажимает ВВОД и CONTROL+D (чтобы пометить конец файла), aspell записывает неправильно написанное слово в стандартный вывод (на экран):

```
$ aspell list
seperate temperatureВВОД
CONTROL+D
separate
```

at

Выполнение команд в указанное время.

at [ключи]время [дата | +приращение]
atq
atrm список_заданий
batch [ключи] [время]

Утилиты at и batch выполняют команды в указанное время. Они получают команды со стандартного ввода или, если используется ключ `-f`, из файла. Команды выполняются в той же среде, что и команда at или batch. Если не было назначено перенаправление, стандартный вывод и стандартная ошибка из команд направляются пользователю, запустившему команду at или batch, по электронной почте. *Задание* — это группа команд, выполняемых за один вызов at. Утилита batch отличается от at тем, что она осуществляет планирование заданий на запуск при малой загруженности центрального процессора в системе.

Утилита atq выводит список заданий, поставленных в очередь; утилита atrm отменяет задания, ожидающие выполнение с помощью утилиты at.

Аргументы

Время — это время суток, когда at запускает задание. Указать ***время*** можно с помощью числа, состоящего из одной, двух или четырех цифр. Числа из одной или двух цифр указывают час, а числа из четырех цифр — час и минуту. ***Время*** можно также задавать в формате чч:мм. Если сразу же за числом не стоит уточнение `am` или `pm` (до или после полудня), при котором используется 12-часовой формат, утилита at предполагает, что время задается в 24-часовом формате. ***Время*** также можно указать как `now` (сейчас), `midnight` (полночь), `noon` (полдень) или `teatime` (время пить чай, 17:00).

Дата — это день недели или день месяца, когда требуется выполнить задание. Если день не указан, at выполняет задание в этот же день, если указанное время больше текущего. Если время меньше текущего, at выполняет задание на следующий день.

День недели указывается его полным названием или аббревиатурой из трех букв. Можно также использовать слова today (сегодня) или tomorrow (завтра). Для указания ***даты*** используется название месяца, за которым следует число дня месяца. За месяцем и числом можно указать год.

Приращение — это число, за которым следует одно из следующих определений (как в единственном, так и во множественном числе): minutes (минуты), hours (часы), days (дни) или weeks (недели). Утилита at добавляет ***приращение*** ко ***времени***. Указать приращение к дате.

При использовании atrm ***списком_заданий*** является список из одного или нескольких номеров заданий. Номера заданий можно указать списком при запуске at с ключом -l или при использовании atq.

Ключи

Утилита batch допускает использование ключей только при работе под управлением OS X. Ключи -c, -d и -l не предназначены для запуска задания с помощью утилиты at, и используются только для определения статуса задания или для прекращения выполнения задания.

`-c` *список_заданий* (cat — связать). Вывод окружения и команд, определенных для заданий, чьи номера входят в *список_заданий*.

`-d` *список_заданий* (delete — удалить). Отмена заданий, запущенных с помощью at. *Список_заданий* — это список, состоящий из одного или нескольких номеров тех заданий, выполнение которых нужно отменить. Если номер задания в памяти не сохранился, следует воспользоваться ключом -l или запустить утилиту atq, чтобы вывести список своих заданий и их номеров. Использование этого ключа с утилитой at приводит к тому же результату, что и запуск утилиты atrm. При работе под управлением OS X этот ключ не используется, вместо него следует использовать ключ -r.

`-f` *файл* (file — файл). Указание, что команды будут взяты из файла, а не со стандартного ввода. Этот ключ пригодится для длинных списков команд или для команд, выполняемых многократно.

`-l` (list — список). Вывод списка ваших заданий для утилиты at. Использование этого ключа с утилитой at приводит к тому же результату, что и запуск утилиты atq.

`-m` (mail — почта). Отправка вам сообщения по электронной почте после запуска задания, даже когда ничего не было отправлено на стандартный вывод или на стандартную ошибку. Когда задание генерирует какой-нибудь вывод, at всегда сообщает вам об этом по электронной почте, независимо от того, используется этот ключ или нет.

`-r` *список_заданий* (remove — удаление). Действие, аналогичное действию ключа -d. (OS X)

Примечания

Утилита at дл выполнения команд использует утилиту /bin/sh. Под управлением Linux этот файл обычно является ссылкой на bash или dash.

Оболочка сохраняет переменные окружения и рабочий каталог на момент передачи задания утилите at, поэтому они доступны при выполнении команд.

at.allow, at.deny

Пользователь, работающий с root-правами, может использовать at в любое время. Кто из обычных, локальных пользователей может использовать at, определяется в файлах Linux /etc/at.allow (OS X использует /var/at/at.allow) и /etc/at.deny (OS X использует /var/at/at.deny), которые должны быть доступны для чтения и записи только пользователям с root-правами (режим 600). Если существует файл at.deny, не имеющий содержимого, использование at доступно всем пользователям. Пользователи, перечисленные в at.deny, не могут пользоваться at, если только они также не входят в список файла at.allow.

При работе под управлением Linux задания, запускаемые с помощью at, выполняются демоном atd. Этот демон хранит задания в файле /var/spool/at или в файле /var/spool/cron/atjobs, а их вывод хранит в файле /var/spool/at/spool или в файле /var/spool/cron/atspool. Эти файлы должны быть настроены на режим 700 и иметь в качестве владельца пользователя по имени daemon.

При работе под управлением Mac OS X задания, запускаемые с помощью at, выполняются утилитой atrun, которая вызывается каждые 30 секунд демоном launchd. Утилита atrun хранит задания в файле /var/at/jobs, а их вывод — в файле /var/at/spool, оба из которых должны быть настроены на режим 700 и иметь в качестве владельца пользователя по имени daemon.

При работе под управлением OS X 10.4 и более поздних версий демон atrun по умолчанию отключен. При работе в качестве привилегированного пользователя atrun можно включать и отключать с помощью следующих команд:

```
# launchctl load -w /System/Library/LaunchDaemons/com.apple.atrun.plist
# launchctl unload -w /System/Library/LaunchDaemons/com.apple.atrun.plist
```

Дополнительные сведения даны в разделе, посвященном демону launchctl.

Примеры

Для разбиения на страницы и распечатки файла long_file завтра в 2 часа ночи (2:00 AM) можно воспользоваться любой из следующих технологий. В первом примере команда выполняется непосредственно из командной строки; в последних двух примерах используется файл pr_tonight, содержащий необходимые команды и выполняющий эти команды с помощью at. Приглашения на ввод данных и вывод из различных версий at отличаются друг от друга.

```
$ at 2am
at> pr long_file | lpr
at>CONTROL+D<EOT>
job 8 at 2009-08-17 02:00

$ cat pr_tonight
#!/bin/bash
pr long_file | lpr
```

```
$ at -f pr_tonight 2am
job 9 at 2009-08-17 02:00

$ at 2am < pr_tonight
job 10 at 2009-08-17 02:00
```

Если команда выполняется непосредственно из командной строки, то конец команды нужно обозначить, нажав в начале новой строки комбинацию CONTROL+D. После нажатия этой комбинации, at выводит строку, которая начинается со слова job (задание), за которым следует номер задания и время, в которое at его выполнит.

Если после предыдущих команд запустить утилиту atq, она выведет список стоящих в очереди заданий:

```
$ atq
8       2009-08-17 02:00 a
9       2009-08-17 02:00 a
10      2009-08-17 02:00 a
```

Следующая команда удаляет из очереди задание номер 9:

```
$ atrm 9
$ atq
8       2009-08-17 02:00 a
10      2009-08-17 02:00 a
```

В следующем примере сценарий cmdfile выполняется в 15 часов 30 минут (3:30 PM) через неделю от текущих суток:

```
$ at -f cmdfile 1530 +1 week
job 12 at 2009-08-23 15:30
```

Следующая команда at выполняет задание в 19:00 (7 PM) в четверг (Thursday). Это задание использует утилиту find для создания промежуточного файла, перенаправления вывода, отправляемого на стандартную ошибку, и распечатки файла.

```
$ at 7pm Thursday
at> find / -name «core" -print >report.out 2>report.err
at> lpr report.out
at>CONTROL+D<EOT>
job 13 at 2009-08-18 19:00
```

В последнем примере показано несколько выводов, сгенерированных благодаря использованию ключа -c после того, как для at была образована очередь из предыдущего задания. В большинстве строк показана среда окружения, и только в последних нескольких строках выполняются команды:

```
$ at -c 13
#!/bin/sh
# atrun uid=500 gid=500
# mail     sam 0
umask 2
PATH=/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:.;
export PATH
PWD=/home/sam/book.examples/99/cp; export PWD
EXINIT=set\ ai\ aw; export EXINIT
LANG=C; export LANG
```

```
PS1=\\\$\ ; export PS1
…
cd /home/sam/book\.examples/99/cp || {
        echo 'Execution directory inaccessible' >&2
        exit 1
}
find / -name "core" -print >report.out 2>report.err
lpr report.out
```

bzip2

Сжатие и восстановление файлов

bzip2 [ключи] [список_файлов]
bunzip2 [ключи] [список_файлов]
bzcat [ключи] [список_файлов]
bzip2recover [файл]

Утилита bzip2 осуществляет сжатие файлов, утилита bunzip2 восстанавливает файлы, сжатые утилитой bzip2, а утилита bzcat выводит содержимое файлов, сжатых утилитой bzip2.

Аргументы

Список_файлов — это список, состоящий из одного или нескольких файлов (не являющихся каталогами), которые нужно сжать или восстановить. Если ***список_файлов*** пуст или присутствует специальный ключ -, bzip2 осуществляет чтение со стандартного ввода. Ключ `--stdout` заставляет bzip2 осуществлять запись на стандартный вывод.

Ключи

При работе под управлением Linux утилиты bzip2, bunzip2 и bzcat допускают применение общих ключей, описание которых дано в табл. V.2.

ВЕРСИЯ BZIP2 ДЛЯ MAC OS X ДОПУСКАЕТ ПРИМЕНЕНИЕ ДЛИННЫХ КЛЮЧЕЙ

Ключи для bzip2, перед которыми ставится двойной дефис (– –), работают как под Mac OS X, так и под Linux.

`--stdout -c` — Запись результатов сжатия или восстановления на стандартный вывод.

`--decompress -d` — Восстановление файла, сжатого с помощью bzip2. Использование утилиты bzip2 с этим ключом эквивалентно использованию команды bunzip2.

`--fast` или `-n` `--best` — Задание размера блока при сжатии файла. При этом *n* может быть цифрой в диапазоне от 1 до 9, где 1 (`--fast`) приводит к созданию блоков размером в 100 Кб, а 9 (`--best`) приводит к созданию блоков размером 900 Кб. По умолчанию используется уровень 9. Ключи `--fast` и `--best` предоставлены для совместимости с утилитой gzip и на самом деле не гарантируют самое быстрое (fast) или самое эффективное (best) сжатие.

`--force -f` Принудительное сжатие, даже если файл уже существует, имеет несколько ссылок или поступает непосредственно из терминала. Такой же эффект этот ключ вызывает при использовании с утилитой bunzip2.

`--keep -k` Отказ от удаления файлов ввода при их сжатии или восстановлении.

`--quiet -q` Подавление предупреждающих сообщений, но вывод важных сообщений.

`--test -t` Проверка целостности сжатого файла. Если файл в порядке, то утилита ничего не выводит.

`--verbose -v` Для каждого сжатого файла вывод имени файла, степени сжатия, процента сэкономленного пространства и размеров несжатого и сжатого файла.

Комментарий

Утилиты bzip2 и bunzip2 работают так же, как и утилиты gzip и gunzip; дополнительные сведения даны в разделе, посвященном gzip. Обычно bzip2 не переписывает файл; чтобы его переписать в процессе сжатия или восстановления, нужно воспользоваться ключом `--force`.

Примечания

Дополнительные сведения об утилите tar и примеры ее использования даны в разделе «tar: Запаковка и распаковка архивов» на с. 82.

Главная страница bzip2 находится на веб-сайте bzip.org.

Утилита bzip2 сжимает файлы эффективнее утилиты gzip.

Чтобы сжимать архивы с помощью bzip2, с утилитой tar используется модификатор `--bzip2`.

`bzcat` *список_файлов* Работает как утилита cat, за исключением того, что использует для восстановления файлов указанный в ***списке_файлов*** bunzip2 по мере их копирования на стандартный вывод.

`bzip2recover` Пытается восстановить поврежденный файл, сжатый утилитой bzip2.

Примеры

В следующих примерах bzip2 сжимает файл и дает созданному файлу такое же имя, но с расширением .bz2. Ключ –v приводит к выводу статистики сжатия.

```
$ ls -l
total 728
-rw-r--r--  1 sam sam 737414 Feb 20 19:05 bigfile
$ bzip2 -v bigfile
  bigfile: 3.926:1, 2.037 bits/byte, 74.53% saved, 737414 in, 187806 out
$ ls -l
total 188
-rw-r--r--  1 sam sam 187806 Feb 20 19:05 bigfile.bz2
```

Затем утилита touch создает файл с таким же именем, которое было у исходного файла; bunzip2 отказывается переписывать файл в процессе восстановления его из файла bigfile.bz2. Ключ `--force` позволяет bunzip2 переписать файл.

```
$ touch bigfile
$ bunzip2 bigfile.bz2
bunzip2: Output file bigfile already exists. Выводимый файл bigfile уже существует.
$ bunzip2 --force bigfile.bz2
$ ls -l
total 728
-rw-r--r--  1 sam sam 737414 Feb 20 19:05 bigfile
```

cal

Вывод календаря.

cal [ключи] [[месяц]год]

Утилита cal выводит календарь.

Аргументы

Аргументы определяют месяц и год, для которых утилита cal выводит календарь. В качестве ***месяца*** используется десятичное целое число в диапазоне от 1 до 12, а ***год*** также указывается в виде десятичного целого числа. В отсутствие аргументов cal выводит календарь на текущий месяц. При указании только одного аргумента он воспринимается как год.

Ключи

`-j` (Julian — юлианский). Вывод юлианского календаря, где дни последовательно отсчитываются с 1 января (1) и до 31 декабря (365 или 366).

`-m` (Monday — понедельник). Превращение понедельника в первый день недели. Без данного ключа первым днем недели будет воскресенье. (Linux)

`-y` (year — год). Вывод календаря для текущего года. (Linux)

`-3` (три месяца). Вывод предыдущего, текущего и следующего месяцев. (Linux)

Примечания

Не сокращайте число года. Указание 05 отличается от указания 2005.

Утилита ncal (new cal — новый календарь) выводит более компактный календарь.

Примеры

Следующая команда выводит календарь на июнь 2011 года:

```
$ cal 6 2011<U>vim:
     June 2011
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
```

```
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```

Следующий пример выводит юлианский календарь на 1949 год:

```
$ cal -j 1949
                                1949
       January                                February
 Su  Mo  Tu  We  Th  Fr  Sa   Su  Mo  Tu  We  Th  Fr  Sa
                      1                  32  33  34  35  36
  2   3   4   5   6   7   8   37  38  39  40  41  42  43
  9  10  11  12  13  14  15   44  45  46  47  48  49  50
 16  17  18  19  20  21  22   51  52  53  54  55  56  57
 23  24  25  26  27  28  29   58  59
 30  31
...
```

cat

Объединение и вывод файлов.

cat [ключи] [список_файлов]

Утилита cat копирует файлы на стандартный вывод. Эту утилиту можно использовать для вывода содержимого одного или нескольких текстовых файлов на экран.

Аргументы

Список_файлов — это список путевых имен одного или нескольких файлов, обрабатываемых утилитой cat. Если аргумент не указан или если вместо имени файла указан дефис (-), cat читает данные со стандартного ввода.

Ключи

Под управлением Linux cat допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--show-all -A`	Работает так же, как ключ `-vET`. (Linux)
`--number-nonblank -b`	Нумерация всех непустых строк по мере их записи на стандартный вывод.
`-e`	(end — и). Работает так же, как ключ `-vE`.
`--show-ends -E`	Пометка конца строк знаком доллара. (Linux)
`--number -n`	Нумерация всех строк по мере их записи на стандартный вывод.
`--squeeze-blank -s`	Удаление лишних пустых строк, чтобы подряд никогда не следовало от двух и более пустых строк.

`-t (tab)` Работает так же, как ключ `-vT`.

`--show-tabs` `-T` Вывод символов табуляции в виде `^I`. (Linux)

`--show-nonprinting` `-v` Вывод управляющих символов в системе записи со знаком вставки (`^M`) и вывод символов с установленным старшим битом (META-символов) в системе записи, использующей префикс `M-` (см. раздел «Клавиши: система записи обозначений и использование» на с. 231). Этот ключ не приводит к преобразованию символов табуляции и перевода строк. Если нужно, чтобы символы табуляции выводились как `^I`, следует воспользоваться ключом `-T` (`--show-tabs`). Символы перевода строк не имеют альтернативного отображения, в противном случае строки были бы слишком длинными.

Примечания

Утилита cat, стандартный ввод и стандартный вывод рассматриваются в разделе «Клавиатура и экран как стандартный ввод и стандартный вывод» на с. 144.

Для вывода содержимого файла, не являющегося текстовым (например, исполняемого программного файла), используется утилита od (см. раздел «od»).

Для вывода строк текстового файла в обратном порядке используется утилита tac. Дополнительные сведения даны на info-странице, посвященной tac.

Имя cat происходит от одной из функций этой утилиты, catenate, что означает последовательное объединение в непрерывную цепь.

ДЛЯ ПРЕДОТВРАЩЕНИЯ ПЕРЕЗАПИСИ ФАЙЛА НУЖНО ВКЛЮЧИТЬ СВОЙСТВО NOCLOBBER

Несмотря на предупреждающие сообщения, выводимые утилитой cat, в следующем примере оболочка перед вызовом cat уничтожает файл ввода (letter):

```
$ cat memo letter > letter
cat: letter: input file is output file   (letter: файл ввода является файлом вывода)
```

Перезапись файла в данной ситуации можно предотвратить, установив значение переменной noclobber (см. главу 5, раздел «noclobber: Предотвращение перезаписи файлов» на с. 148 и главу 9, табл. 9.7).

Примеры

Следующая команда выводит на терминал содержимое текстового файла memo:

```
$ cat memo
...
```

В следующем примере происходит объединение трех текстовых файлов и перенаправление вывода в файл all:

```
$ cat page1 letter memo > all
```

Утилиту cat можно использовать для создания коротких текстовых файлов без использования редактора. Введите следующую командную строку, наберите текст, который нужно поместить в файл, и в отдельной строке нажмите CONTROL+D:

```
$ cat > new_file
...
```

```
(текст)
...
CONTROL+D
```

В данном случае cat получает ввод со стандартного ввода (клавиатуры), а оболочка перенаправляет стандартный вывод (копию ввода) в указанный вами файл. Комбинация CONTROL+D дает сигнал о конце файла — EOF (end of file) и заставляет cat вернуть управление оболочке.

В следующем примере канал отправляет вывод команды who на стандартный ввод утилиты cat. Оболочка перенаправляет вывод утилиты cat в файл по имени output; после того как выполнение команды будет завершено, содержимым output станет содержимое файла header (заголовок), вывод команды who и содержимое файла footer (нижняя сноска). Дефис в командной строке заставляет cat прочитать после файла header и до чтения файла footer данные со стандартного ввода.

```
$ who | cat header - footer > output
```

cd

Смена рабочего каталога.

cd [ключи] [каталог]

Встроенная команда cd делает ***каталог*** рабочим.

Аргументы

Каталог — это путевое имя каталога, который нужно сделать новым рабочим каталогом. Без аргумента cd делает рабочим ваш личный каталог. Использование дефиса вместо ***каталога*** снова делает рабочим предыдущий рабочий каталог.

Ключи

Следующие ключи допускаются только при работе под оболочками bash и dash.

`-L` (без разыменования). Если каталог является символьной ссылкой, cd превращает символьную ссылку в рабочий каталог (по умолчанию).

`-P` (с разыменованием). Если каталог является символьной ссылкой, cd превращает каталог, на который указывает символьная ссылка, в рабочий.

Примечания

Команда cd является встроенной командой оболочек bash, dash и tcsh.

Команда cd подробно рассмотрена в главе 4, в разделе «cd: Смена рабочего каталога» на с. 108.

При отсутствии аргументов cd делает рабочим ваш личный каталог; чтобы опередить путевое имя личного каталога, она использует значение переменной HOME (bash; см. раздел «Переменная HOME: Ваш личный каталог» на с. 311) или переменной home (tcsh; см. раздел «Переменные оболочки, принимающие значения», подраздел «home или HOME» на с. 391).

Когда в качестве аргумента используется дефис, cd снова делает рабочим предыдущий рабочий каталог. Для определения путевого имени предыдущего рабочего каталога она использует значение переменной OLDPWD (bash) или переменной owd (tcsh).

Переменная CDPATH (bash) или переменная cdpath (tcsh) содержит список каталогов, разделенных двоеточиями, в котором утилита cd ведет поиск. В этом списке нулевое имя каталога (::) или точка (:.:) означает рабочий каталог. Если значение переменной CDPATH или переменной cdpath не установлено, cd ведет поиск ***каталога*** только в рабочем каталоге. Если значение этой переменной установлено и ***каталог*** не является абсолютным путевым именем (не начинается с символа слэша), cd ведет поиск среди каталогов списка; если поиск заканчивается безрезультатно, cd ведет поиск в рабочем каталоге. Переменная CDPATH рассматривается в главе 8, в разделе «Переменная CDPATH: Расширение возможностей команды cd».

Примеры

Команда cd без аргументов делает рабочим личный каталог пользователя. В следующем примере cd делает рабочим личный каталог пользователя Max, и это изменение подтверждается встроенной командой pwd.

```
$ pwd
/home/max/literature
$ cd
$ pwd
/home/max
```

При работе под управлением Mac OS X личные каталоги хранятся в каталоге /Users, а не в каталоге /home.

Следующая команда делает рабочим каталог /home/max/literature:

```
$ cd /home/max/literature
$ pwd
/home/max/literature
```

Затем утилита cd делает новым рабочим каталогом подкаталог рабочего каталога:

```
$ cd memos
$ pwd
/home/max/literature/memos
```

И наконец, утилита cd использует ссылку .. на родительский каталог (см. раздел «Элементы каталога . и ..» на с. 108), чтобы этот родительский каталог стал новым рабочим каталогом:

```
$ cd ..
$ pwd
/home/max/literature
```

chgrp

Изменение группы, связанной с файлом.

chgrp [ключи] группа список_файлов
chgrp [ключи] — –reference=rfile список_файлов (Linux)

Утилита chgrp изменяет группу, связанную с одним или несколькими файлами. Второй формат работает только под управлением Linux.

Аргументы

Группа — это имя или числовой идентификатор (ID) новой группы. ***Список_файлов*** — это список путевых имен файлов, чья групповая связь должна быть изменена. Аргумент ***rfile*** — это путевое имя файла, чья группа должна стать новой группой, связанной с файлами, перечисленными в ***списке_файлов***.

Ключи

Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--changes -c` Вывод сообщения для каждого файла, чья группа изменяется. (Linux)

`--dereference` Для каждого файла, являющегося символьной ссылкой, изменение группы файла, на которую указывает ссылка, а не самой символьной ссылки. При работе под управлением Linux этот ключ используется по умолчанию. (Linux)

`--quiet -f --silent` Подавление предупреждающих сообщений о тех файлах, права доступа к которым препятствуют в изменении их ID группы.

`--no-dereference -h` Для каждого файла, являющегося символьной ссылкой, изменение группы символьной ссылки, а не файла, на который эта ссылка указывает.

`-H` (с частичным разыменованием). Для каждого файла, являющегося символьной ссылкой, изменение группы файла, на который указывает ссылка, а не самой символьной ссылки. Действие этого ключа распространяется на файлы, указанные в командной строке, оно не касается файлов, найденных при спуске по иерархии каталога. Этот ключ работает только с ключом `-R`. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «Сравнение ключей –H и –L» на с. 638.

`-L` (с разыменованием). Для каждого файла, являющегося символьной ссылкой, изменение группы файла, на который указывает ссылка, а не самой символьной ссылки. Действие этого ключа распространяется на все файлы. Он работает только с ключом `-R`. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «Сравнение ключей –H и –L» на с. 638.

`-P` (без разыменования). Для каждого файла, являющегося символьной ссылкой, изменение группы символьной ссылки, а не файла, на который эта ссылка указывает (используется по умолчанию). Этот ключ воздействует на все файлы и работает только с ключом `-R`. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в подразделе «-P» раздела «Примеры» на с. 640.

`--recursive` `-R` — Рекурсивный спуск по каталогу, указанному в ***списке_файлов***, и изменение ID группы всех файлов в иерархии каталога.

`--reference=`***`rfile`*** — Изменение группы для файлов, указанных в ***списке_файлов***, на группу файла ***rfile***. (Linux)

`--verbose -v` — Вывод сообщения для каждого файла о сохранении или изменении группы, с которой он связан.

Примечания

Группу, связанную с файлом, может изменить только владелец файла или пользователь с root-правами.

Чтобы при работе без root-прав изменить для файла ID группы на идентификатор какой-нибудь другой группы, нужно этой группе принадлежать.

Обратите также внимание на раздел, посвященный утилите chown, где даны сведения о том, как изменить группу, связанную с файлом, а также владельца файла с помощью этой утилиты.

Примеры

Следующая команда изменяет группу, с которой связан файл manuals, на новую группу pubs.

```
$ chgrp pubs manuals
```

Сравнение ключей -H и -L — В следующих примерах показывается разница в работе утилиты с ключом -H и с ключом -L. Изначально все файлы и каталоги в рабочем каталоге связаны с группой zach:

```
$ ls -lR
.:
total 12
-rw-r--r-- 1 zach zach  102 Jul  2 12:31 bb
drwxr-xr-x 2 zach zach 4096 Jul  2 15:34 dir1
drwxr-xr-x 2 zach zach 4096 Jul  2 15:33 dir4

./dir1:
total 4
-rw-r--r-- 1 zach zach 102 Jul  2 12:32 dd
lrwxrwxrwx 1 zach zach   7 Jul  2 15:33 dir4.link -> ../dir4

./dir4:
total 8
-rw-r--r-- 1 zach zach 125 Jul  2 15:33 gg
-rw-r--r-- 1 zach zach 375 Jul  2 15:33 hh
```

При вызове утилиты chgrp с ключами -R и -H (ключ -H без ключа -R не работает) chgrp разыменовывает только те символьные ссылки, которые перечислены в командной строке, куда включаются и символьные ссылки, найденные в каталогах, перечисленных в командной строке. Это означает, что chgrp изменяет связь с группой для тех файлов, на которые указывают ссылки. Утилита не разыменовывает символьные

ссылки, найденные по мере спуска по иерархиям каталогов, и не вносит изменения в сами символьные ссылки. При спуске по иерархии каталога dir1 утилита chgrp не изменяет dir4.link, но изменяет dir4, то есть тот каталог, на который указывает dir4.link.

```
$ chgrp -RH pubs bb dir1
$ ls -lR
.:
total 12
-rw-r--r-- 1 zach pubs  102 Jul  2 12:31 bb
drwxr-xr-x 2 zach pubs 4096 Jul  2 15:34 dir1
drwxr-xr-x 2 zach pubs 4096 Jul  2 15:33 dir4

./dir1:
total 4
-rw-r--r-- 1 zach pubs 102 Jul  2 12:32 dd
lrwxrwxrwx 1 zach zach   7 Jul  2 15:33 dir4.link -> ../dir4

./dir4:
total 8
-rw-r--r-- 1 zach zach 125 Jul  2 15:33 gg
-rw-r--r-- 1 zach zach 375 Jul  2 15:33 hh
```

ДЕЙСТВИЕ КЛЮЧА –H ПРИ РАБОТЕ В MAC OS X

Под управлением Mac OS X ключ –H утилиты chgrp оказывает на ее работу несколько иное действие, чем при работе под управлением Linux. Под управлением OS X, команда chgrp –RH изменяет группу символьной ссылки, найденной в каталоге, перечисленном в командной строке, и не изменяет файл, на который указывает эта ссылка. (Он не приводит к разыменованию символьной ссылки.) При запуске предыдущего примера под управлением OS X привязка к группе файлов каталога dir4 не изменится, а будет изменена привязка к группе файла dir4.link.

Если работа вашей программы зависит от функционирования утилиты с ключом –H под управление OS X, следует протестировать ключ с этой утилитой, чтобы конкретно определить, как она с ним работает.

При вызове утилиты chgrp с ключами -R и -L (ключ -L без ключа -R не работает) chgrp разыменовывает все символьные ссылки: и те, что перечислены в командной строке, и те, что найдены при спуске по иерархии каталогов. Сами символьные ссылки при этом не изменяются. Следующая команда изменяет файлы, на которые указывает ссылка dir4.link:

```
$ chgrp -RL pubs bb dir1

$ ls -lR
.:
total 12
-rw-r--r-- 1 zach pubs  102 Jul  2 12:31 bb
drwxr-xr-x 2 zach pubs 4096 Jul  2 15:34 dir1
drwxr-xr-x 2 zach pubs 4096 Jul  2 15:33 dir4

./dir1:
total 4
-rw-r--r-- 1 zach pubs 102 Jul  2 12:32 dd
```

```
lrwxrwxrwx 1 zach zach   7 Jul  2 15:33 dir4.link -> ../dir4
./dir4:
total 8
-rw-r--r-- 1 zach pubs 125 Jul  2 15:33 gg
-rw-r--r-- 1 zach pubs 375 Jul  2 15:33 hh
```

`-P` При вызове утилиты chgrp с ключами `-R` и `-P` (ключ `-P` без ключа `-R` не работает) chgrp не разыменовывает символьные ссылки. Она изменяет группу, связанную с самими символьными ссылками.

```
$ ls -l bb*
-rw-r--r-- 1 zach zach 102 Jul  2 12:31 bb
lrwxrwxrwx 1 zach zach   2 Jul  2 16:02 bb.link -> bb

$ chgrp -PR pubs bb.link
$ ls -l bb*
-rw-r--r-- 1 zach zach 102 Jul  2 12:31 bb
lrwxrwxrwx 1 zach pubs   2 Jul  2 16:02 bb.link -> bb
```

chmod

Изменение режима (прав) доступа к файлу.

chmod [ключи] кто оператор право список_файлов	символьный
chmod [ключи] режим список_файлов	абсолютный
chmod [ключи] — –reference=rfile список_файлов	ссылочный (Linux)

Утилита chmod изменяет способы доступа к файлу со стороны владельца файла, группы, с которой связан файл, и (или) всех остальных пользователей. Новый режим доступа можно указать абсолютным или символьным способом. При работе под управлением Linux можно также указать режим ссылочным способом (третий формат). При работе под управлением Mac OS X утилиту chmod можно использовать для изменения ACL-списков.

Аргументы

Аргументы указывают, к каким файлам и какими способами должны измениться режимы доступа. Аргумент ***rfile*** является путевым именем файла, права доступа к которому станут новыми правами доступа для файлов, перечисленных в ***списке_файлов***.

Символьный режим

В символьном режиме можно указать сразу несколько наборов установок (***кто оператор право***), отделяя каждый набор от следующего запятой.

Утилита chmod изменяет права доступа для класса пользователей, указанного в аргументе ***кто***. Класс пользователей определяется одной или несколькими буквами, указанными в столбце ***кто*** в табл. V.4.

В табл. V.5 перечислены операторы, используемые в символьном режиме.

Права доступа указываются одной или несколькими буквами, перечисленными в табл. V.6.

Таблица V.4. Указание класса пользователей в символьном режиме

кто	Класс пользователей	Значение
u	Пользователь	Владелец файла
g	Группа	Группа, с которой связан файл
o	Все остальные	Все остальные пользователи
a	Все	Используется вместо ugo

Таблица V.5. Операторы в символьном режиме

оператор	Значение
+	Добавление права для указанного класса пользователей
-	Удаление права для указанного класса пользователей
=	Установка права для указанного класса пользователей; переустанавливает все остальные права для данного класса пользователей

Таблица V.6. Права в символьном режиме

право	Значение
r	Установка права на чтение
w	Установка права на запись
x	Установка права на выполнение
s	Установка ID пользователя или ID группы (в зависимости от содержимого аргумента ***кто***) в качестве владельца выполняемого файла. (Дополнительные сведения даны в главе 4, в разделе «Права setuid и setgid» на с. 118.)
t	Установка бита флажка ограничения прав на удаление — sticky bit. (Это право может быть установлено только владельцем, работающим с root-правами, и может использоваться только с аргументом ***кто***, имеющим значение u)
X	Превращение файла в исполняемый, только если он является каталогом или если у класса пользователей «все остальные» есть права на выполнение
u	Установка таких же прав, как у владельца
g	Установка таких же прав, как у группы
o	Установка таких же прав, как у всех остальных

Абсолютный режим

Для указания режима нужно использовать восьмеричное число. Число составляется путем обработки операцией ИЛИ значений из табл. V.7. Чтобы получить результат обработки двух и более восьмеричных чисел из этой таблицы операцией ИЛИ (OR), их нужно просто сложить. (Примеры показаны в табл. V.8.)

Таблица V.7. Указатели абсолютного режима

режим	Значение
4000	Установка ID пользователя при выполнении программы (раздел «Права setuid и setgid» на с. 118)
2000	Установка ID группы при выполнении программы (раздел «Права setuid и setgid)

продолжение ⇩

Таблица V.7 (*продолжение*)

режим	Значение
1000	Бит флажка ограничения прав на удаление (sticky bit)
0400	Присваивание владельцу права на чтение файла
0200	Присваивание владельцу права на запись файла
0100	Присваивание владельцу права на выполнение файла
0040	Присваивание группе права на чтение файла
0020	Присваивание группе права на запись файла
0010	Присваивание группе права на выполнение файла
0004	Присваивание всем остальным права на чтение файла
0002	Присваивание всем остальным права на запись файла
0001	Присваивание всем остальным права на выполнение файла

В табл. V.8 перечислены некоторые типовые ***режимы***.

Таблица V.8. Примеры указателей абсолютного режима

режим	Значение
0777	Установка прав на чтение, запись и выполнение файла для владельца, группы и всех остальных
0755	Установка для владельца прав на чтение, запись и выполнение файла, установка для группы и для всех остальных прав на чтение и выполнение файла
0711	Установка для владельца прав на чтение, запись и выполнение файла, установка для группы и для всех остальных права на выполнение файла
0644	Установка для владельца прав на чтение и запись файла, установка для группы и для всех остальных права на чтение файла
0640	Установка для владельца прав на чтение и запись файла, установка для группы права на чтение файла и запрещение всем остальным доступа к файлу

Ключи

Ключи, перед которыми ставится двойной дефис (`--`), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--changes -c` Вывод сообщения для каждого файла, права доступа к которому изменяются. (Linux)

`--quiet -f --silent` Подавление предупредительных сообщений о файлах, права доступа к которым препятствуют chmod в изменении прав доступа к файлу.

`-H` (с частичным разыменованием). Для каждого файла, являющегося символьной ссылкой, изменение прав доступа к файлу, на который указывает ссылка, а не к самой символьной ссылке. Действие этого ключа распространяется на файлы, указанные в командной строке, оно не касается файлов, найденных при спуске по иерархии каталога. Этот ключ работает только с ключом `-R`. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом.

Пример его использования приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L». (OS X)

`-L` (с разыменованием). Для каждого файла, являющегося символьной ссылкой, изменение прав доступа к файлу, на который указывает ссылка, а не к самой символьной ссылке. Действие этого ключа распространяется на все файлы. Он работает только с ключом `-R`. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L». (OS X)

`-P` (без разыменования). Для каждого файла, являющегося символьной ссылкой, изменение прав доступа к символьной ссылке, а не к файлу, на который эта ссылка указывает (используется по умолчанию). Действие этого ключа распространяется на все файлы. Он работает только с ключом `-R`. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «chgrp», в подразделе «Примеры». (OS X)

`--recursive` `-R` Рекурсивный спуск по каталогу, указанному в ***списке_файлов***, и изменение прав доступа к файлам иерархии каталога.

`--reference=`*`rfile`* Изменение прав доступа к файлам, указанным в ***списке_файлов*** на права доступа, которые имеются к файлу ***rfile***. (Linux)

`--verbose` `-v` Вывод сообщения для каждого файла о том, что права доступа к нему были изменены (даже если они не были изменены), и указание прав доступа. Для вывода сообщений только в том случае, когда права действительно были изменены, используется ключ `--changes`.

Примечания

Изменить права (или режим) доступа к файлу может только владелец файла или пользователь, работающий с root-правами.

Если с символьными аргументами используется ***оператор*** =, то ***право*** можно опустить, что приводит к удалению всех прав для указанного класса пользователей. См. второй пример в следующем разделе.

При работе под управлением Linux утилита chmod никогда не изменяет права доступа к символьным ссылкам.

Под Linux chmod разыменовывает символьные ссылки, найденные в командной строке. Иными словами, chmod изменяет права доступа к файлам, на которые указывают символьные ссылки, найденные в командной строке, и не затрагивает файлы, найденные при спуске по иерархии каталога. Этот поведение похоже на поведение утилиты при работе по управлением Mac OS X с ключом `-H`.

Утилита chmod рассмотрена также в разделе «chmod: Изменение прав доступа» на с. 116.

Примеры

В следующем примере показано использование утилиты chmod для изменения прав доступа к файлу по имени temp. Исходные права доступа к файлу temp показаны

утилитой ls. Сведения о том, что выводится с помощью утилиты ls, даны в посвященном ей разделе, в подразделе «Комментарии».

```
$ ls -l temp
-rw-rw-r-- 1 max pubs 57 Jul 12 16:47 temp
```

Когда за знаком равенства не указываются ***право***, chmod удаляет все права для указанного класса пользователей. Следующая команда удаляет все права доступа для всех остальных пользователей, оставляя право доступа к файлу только для владельца:

```
$ chmod go= temp
$ ls -l temp
-rw------- 1 max pubs 57 Jul 12 16:47 temp
```

Следующая команда изменяет режимы доступа для всех пользователей (владельца, группы и всех остальных), давая им право на чтение файла и запись в него.

```
$ chmod a=rw temp
$ ls -l temp
-rw-rw-rw- 1 max pubs 57 Jul 12 16:47 temp
```

При использовании абсолютного аргумента a=rw превращается в 666. Следующая команда выполняет ту же функцию, что и предыдущая:

```
$ chmod 666 temp
```

Следующая команда удаляет права на запись для всех остальных пользователей. В результате этого участники группы pubs могут осуществлять чтение из файла и запись в него, но все остальные пользователи смогут только читать файл:

```
$ chmod o-w temp
$ ls -l temp
-rw-rw-r-- 1 max pubs 57 Jul 12 16:47 temp
```

Следующая команда приводит к такому же результату с использованием абсолютного аргумента:

```
$ chmod 664 temp
```

Следующая команда добавляет право на выполнение файла всем пользователям:

```
$ chmod a+x temp
$ ls -l temp
-rwxrwxr-x 1 max pubs 57 Jul 12 16:47 temp
```

Если файл temp является сценарием оболочки или другим исполняемым файлом, то теперь его смогут запустить на выполнение все пользователи. (Операционная система требует для выполнения сценария оболочки прав на чтение и выполнение его файла, а для выполнения двоичного файла — только права на его выполнение.) Абсолютная команда, приводящая к такому же результату, имеет следующий вид:

```
$ chmod 775 temp
```

В последней команде для достижения такого же результата, что и в предыдущей, используются символьные аргументы. Она устанавливает права на чтение, запись и выполнение для владельца и группы и права на чтение и выполнение для всех остальных пользователей. Наборы символьных режимов разделены запятой.

```
$ chmod ug=rwx,o=rx temp
```

chown

Изменение владельца файла и (или) группы, с которой связан файл.

*chown [ключи] **владелец список_файлов***
*chown [ключи] **владелец:группа список_файлов***
*chown [ключи] **владелец: список_файлов***
*chown [ключи] **: группа список_файлов***
*chown [ключи] — –reference=**rfile список_файлов*** (Linux)

Утилита chown изменяет владельца файла и (или) группу, с которой этот файл связан. Владельца файла может изменить только пользователь, работающий с root-правами. Группу, с которой связан файл, может изменить только пользователь, работающий с root-правами, или пользователь файла, принадлежащий новой группе. Последний формат работает только в Linux.

Аргументы

Владелец — это имя нового пользователя или его числовой ID. ***Список_файлов*** — это список путевых имен файлов, чью принадлежность к пользователю или привязку к группе нужно изменить. ***Группа*** — это имя новой группы, с которой должен быть связан файл или ее числовой ID. Аргумент ***rfile*** является путевым именем файла, чей владелец и (или) привязка к группе станут новым владельцем и (или) новой группой, связанной со ***списком_файлов***. В табл. V.9 показаны способы, позволяющие указать нового владельца и (или) группу.

Таблица V.9. Указание нового владельца и (или) группы

Аргумент	Значение
владелец	Новый владелец файлов, указанных в ***списке_файлов***; группа не изменяется
владелец:группа	Новый владелец и новая группа, связанная с файлами, указанными в ***списке_файлов***
владелец:	Новый владелец файлов, указанных в ***списке_файлов***; группа, связанная с файлами, указанными в ***списке_файлов***, изменяется на группу нового владельца, участником которой он зарегистрирован в системе
: группа	Новая группа, связанная с файлами, указанными в ***списке_файлов***; владелец не изменяется

Ключи

При работе под управлением Linux chown допускают применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

--changes -c Вывод сообщения для каждого файла, у которого изменяется владелец или связанная с ним группа. (Linux)

`--dereference` Изменение владельца или связанной группы не для самих символьных ссылок, а для тех файлов, на которые они указывают. При работе под управлением Linux этот ключ используется по умолчанию. (Linux)

`--quiet -f --silent` Подавление сообщений об ошибках для тех файлов, чьих владельцев или связь с группой chown не может изменить.

`-H` (с частичным разыменованием). Для каждого файла, являющегося символьной ссылкой, изменение владельца и (или) связанной с ним группы не для самой ссылки, а для того файла, на который указывает эта ссылка. Действие этого ключа распространяется на файлы, указанные в командной строке, оно не касается файлов, найденных при спуске по иерархии каталога. Этот ключ работает только с ключом `-R`. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L».

`--no-dereference -h` Для каждого файла, являющегося символьной ссылкой, изменение владельца и (или) группы, связанной с символьной ссылкой, а не с файлом, на который она указывает.

`-L` (с разыменованием). Для каждого файла, являющегося символьной ссылкой, изменение владельца и (или) группы, связанной с файлом, не для ссылки, а для файла, на который она указывает. Действие этого ключа распространяется на все файлы. Он работает только с ключом `-R`. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L».

`-P` (без разыменования). Для каждого файла, являющегося символьной ссылкой, изменение владельца и (или) группы, связанной с символьной ссылкой, а не с файлом, на который она указывает. Действие этого ключа распространяется на все файлы. Он работает только с ключом `-R`. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «chgrp», в подразделе «Примеры».

`--recursive -R` Когда в ***список_файлов*** включены каталоги, этот ключ задает рекурсивный спуск по каталогу, устанавливая указанного владельца и (или) группу, связанную со всеми файлами в данной иерархии.

`--reference=`*`rfile`* Изменение владельца и (или) группы, связанной с файлами в ***списке_файлов*** на владельца и (или) группу, связанную с файлом по имени ***rfile***. (Linux)

`--verbose -v` Вывод для каждого файла сообщения о том, что его владелец и (или) группа, с которой он связан, были сохранены или изменены.

Примечания

Утилита chown при изменении владельца файла снимает установку битов `setuid` и `setgid`.

Примеры

Следующая команда изменяет владельца файла chapter1 в каталоге manuals. Новым владельцем становится Sam.

```
# chown sam manuals/chapter1
```

Следующая команда делает пользователя Max владельцем, а группу, в которой он зарегистрирован, группой, связанной со всеми файлами в каталоге /home/max/literature и во всех его подкаталогах.

```
# chown -R max: /home/max/literature
```

При работе под управлением Mac OS X личные каталоги хранятся не в каталоге /home, а в каталоге /Users.

Следующая команда изменяет принадлежность к владельцу файлов в каталоге literature на max, а группу, связанную с этими файлами, на pubs:

```
# chown max:pubs /home/max/literature/*
```

В последнем примере изменяется группа, связанная с файлами в каталоге manuals, на pubs без изменения их принадлежности к владельцу. Владелец файлов, выполняющий эту команду, должен принадлежать к группе pubs.

```
$ chown :pubs manuals/*
```

cmp

Сравнение двух файлов.

cmp [ключи] файл1 [файл2 [пропуск1 [пропуск2]]]

Утилита cmp выводит разницу между двумя файлами на основе их побайтового сравнения. Если файлы одинаковы, cmp ничего не выводит. Если файлы разные, cmp выводит байт и номер строки первого различия.

Аргументы

Аргументы ***файл1*** и ***файл2*** являются путевыми именами файлов, сравниваемых с помощью утилиты cmp. Если ***файл2*** опущен, cmp использует вместо него стандартный ввод. Использование дефиса (-) вместо ***файла1*** или ***файла2*** заставляет cmp вместо использования этого файла осуществлять чтение со стандартного ввода.

Аргументы ***пропуск1*** и ***пропуск2*** являются десятичными числами, указывающими на то количество байтов, которое нужно пропустить в каждом файле перед началом сравнения. После ***пропуск1*** и ***пропуск2*** можно использовать стандартные мультипликативные суффиксы (см. табл. V.1).

Ключи

При работе под управлением Linux и OS X утилита cmp допускает применение общих ключей, показанных в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--print-bytes` `-b` — Вывод информации, включая имена файлов, номера байта и строки, а также восьмеричного и ASCII-значения первого отличающегося байта.

`--ignore-initial=`*n1*`[:`*n2*`]` `-i` *n1*`[:`*n2*`]` — В отсутствие ***n2***, пропуск первых ***n1*** байт в обоих файлах перед началом сравнения. При указании ***n1*** и ***n2***, пропуск перед началом сравнения первых ***n1*** байт в ***файле1*** и первых ***n2*** байт в ***файле2***. За ***n1*** и (или) ***n2*** можно ставить мультипликативные суффиксы (см. табл. V.1).

`--verbose` `-l` (буква «l» в нижнем регистре). Вместо остановки на первом отличающемся байте продолжение сравнивания двух файлов и вывод как местоположения, так и значения каждого отличающегося байта. Местоположения выводятся в виде десятичного байтового счетчика смещения относительно начала файлов, а значения байтов выводятся в восьмеричном виде. Сравнение завершается, когда в любом из файлов будет достигнут конец файла — EOF.

`--silent` *или* `--quiet` `-s` — Подавление вывода из cmp, установка только статуса выхода (см. «Примечания»).

Примечания

Подсчет байтов и строк начинается с 1.

Утилита cmp не выводит сообщение, если файлы идентичны друг другу, она устанавливает лишь статус выхода. Эта утилита возвращает статус выхода 0, если файлы одинаковы, и статус выхода 1, если они различаются. Статус выхода больше единицы означает возникновение ошибки.

При использовании аргумента ***пропуск1*** (и ***пропуск2***) выводимые cmp значения смещения основаны на байте, с которого начинается сравнение.

При работе под управлением Mac OS X cmp сравнивает ветви данных файла (data forks). Если нужно сравнить ветви ресурсов (resource forks), можно вручную сравнить файлы ..namedfork/rsrc с целевыми файлами.

В отличие от утилиты diff (см. соответствующий раздел), cmp работает как с двоичными файлами, так и с файлами ASCII.

Примеры

В показанных ниже примерах используются файлы a и b. Эти файлы различаются в двух местах. Первое различие состоит в том, что слово `lazy` в файле a представлено в файле b как `lasy`. Второе различие имеет более тонкий характер: в файле b непосредственно перед разделителем строк появляется символ табуляции (TAB).

```
$ cat a
The quick brown fox jumped over the lazy dog.
$ cat b
The quick brown fox jumped over the lasy dog.TAB
```

В первом примере утилита cmp используется для сравнения двух файлов без ключей. Она сообщает о том, что файлы различаются, и определяет смещение от начала файлов того места, где находится первое различие:

```
$ cmp a b
a b differ: char 39, line 1    различие: символ 39, строка 1
```

Добавив к команде ключ -b (--print-bytes), можно вывести байты, находящиеся в этом месте:

```
$ cmp --print-bytes a b
a b differ: char 39, line 1 is 172 z 163 s
                                различие: символ 39, строка 1 – 172 z 163 s
```

Ключ -l заставляет вывести все байты, создающие различие между этими двумя файлами. Поскольку этот ключ при множестве различий приводит к выводу большого объема данных, может появиться потребность перенаправить вывод в файл. В следующем примере показаны два различия, имеющиеся между файлами a и b. Ключ -b приводит также к выводу значений для байтов. Там, где в файле a имеется разделитель строк (CONTROL+J), в файле b имеется символ табуляции (CONTROL+I). Сообщение, свидетельствующее о достижении конца файла, показывает, что файл b длиннее файла a.

```
$ cmp -lb a b
39 172 z    163 s
46  12 ^J    11 ^I
cmp: EOF on a
```

В следующем примере для перехода через первое различие в файлах используется ключ --ignore-initial. Теперь утилита cmp сообщает о втором различии. Различие дается в символе 7, который является 46-м символом в исходном файле b (7 символов после проигнорированных 39 символов).

```
$ cmp --ignore-initial=39 a b
a b differ: char 7, line 1
```

В предыдущем примере вместо ключа --ignore-initial можно использовать аргументы ***пропуск1*** и ***пропуск2***:

```
$ cmp a b 39 39
a b differ: char 7, line 1
```

comm

Сравнение отсортированных файлов.

*comm [ключи] **файл1** **файл2***

Утилита comm выводит результат построчного сравнения двух отсортированных файлов. В первом из трех отображаемых столбцов выводится список строк, найденных только в файле с именем ***файл1***, во втором — список строк, найденных только в файле с именем ***файл2***, а в третьем выводится список строк, общих для обоих файлов.

Аргументы

Аргументы ***файл1*** и ***файл2*** — это путевые имена файлов, сравниваемых с помощью утилиты comm. Использование дефиса (-) вместо аргумента ***файл1*** или аргумента ***файл2*** заставляет comm осуществлять чтение не из соответствующего файла, а со стандартного ввода.

Ключи

Ключи можно комбинировать. Без ключей утилита comm выдает вывод, состоящий из трех столбцов.

-1 Отмена вывода первого столбца (отмена вывода строк, найденных только в файле ***файл1***).

-2 Отмена вывода второго столбца (отмена вывода строк, найденных только в файле ***файл2***).

-3 Отмена вывода третьего столбца (отмена вывода строк, найденных в обоих файлах).

Примечания

Если файлы не были отсортированы, comm будет работать неправильно. Строки во втором столбце предваряются одним, а строки в третьем столбце — двумя символами табуляции. Статус выхода свидетельствует о нормальном (0) или ненормальном завершении работы утилиты comm (не 0).

Примеры

В следующих примерах используются два файла: c и d. Как и весь ввод в comm, эти файлы уже отсортированы:

```
$ cat c
bbbbb
ccccc
ddddd
eeeee
fffff

$ cat d
aaaaa
ddddd
eeeee
ggggg
hhhhh
```

Сведения о сортировке файлов даны в разделе, посвященном утилите sort.

В следующем примере утилита comm вызывается без ключей, поэтому она выводит три столбца. В первом столбце выводится список тех строк, которые найдены только в файле c, во втором — список только тех строк, которые найдены в файле d, а в третьем — список строк, найденных в обоих файлах, в c и d:

```
$ comm c d
        aaaaa
bbbbb
ccccc
                ddddd
                eeeee
fffff
        ggggg
        hhhhh
```

В следующем примере показано использование ключей, подавляющих вывод столбцов 1 и 2. В результате выводится столбец 3, список строк, общих для файлов c и d:

```
$ comm -12 c d
ddddd
eeeee
```

configure

Автоматическая конфигурация исходного кода.

*./configure [**ключи**]*

Сценарий configure является частью GNU-системы конфигурации и сборки. Разработчики программного обеспечения, поставляющие исходный код для своих продуктов, столкнулись с задачей упрощения для особо неискушенных пользователей сборки и установки своих программных пакетов на широкую линейку машинных архитектур, операционных систем и системного программного обеспечения. Для облегчения этого процесса многие разработчики программ предоставили вместе со своим исходным кодом сценарий оболочки по имени configure.

При запуске сценария configure он проверяет локальную систему на совместимость с пакетом. Данные, собранные сценарием, используются для создания файлов, с помощью которых утилита make (см. соответствующий раздел) производит сборку исполняемых файлов и библиотек. Поведение сценария configure можно отрегулировать путем указания ключей командной строки и использования переменных окружения.

Ключи

ВЕРСИЯ СЦЕНАРИЯ CONFIGURE ДЛЯ MAC OS X ДОПУСКАЕТ ПРИМЕНЕНИЕ ТОЛЬКО ДЛИННЫХ КЛЮЧЕЙ

Ключи для сценария configure, перед которыми ставится двойной дефис (– –), работают как под управлением Mac OS X, так и под управлением Linux.

`--disable-`*функция* Работает так же, как и `--enable-`***функция***, за исключением того, что этот ключ отключает поддержку ***функции***.

`--enable-`*функция* ***Функция*** — это имя функции, которая может быть поддержана конфигурируемым программным обеспечением. Например, конфигурирование исходного кода Z Shell с использованием команды конфигурации `configure --enable-zsh-mem` настраивает исходный код на использование специального распределения памяти, предоставляемого zsh, вместо использования системных подпрограмм распределения памяти. Чтобы просмотреть варианты, доступные для той или иной функции, нужно прочитать файл README, поставляемый с дистрибуционным пакетом программного обеспечения.

`--help` Вывод подробного списка всех ключей, используемых со сценарием configure. Содержимое этого списка зависит от устанавливаемого программного обеспечения.

`--prefix=`*каталог* По умолчанию сценарий configure создает файлы для сборщика, устанавливающего программное обеспечение, в иерархии каталогов /usr/

local (когда дается команда `make install`). Для установки программного обеспечения в другой каталог замените ***каталог*** путевым именем нужного вам каталога.

`--with-`*пакет* ***Пакет*** — это имя дополнительного пакета, который может быть включен в конфигурируемое программное обеспечение. Например, если с помощью команды `configure --with-dll` конфигурируется исходный код для эмулятора Windows wine, то этот код настраивается на сборку общей библиотеки, поддерживающей эмуляцию Windows. Сведения о том, что можно выбрать в качестве ***пакета***, содержатся в файле README, поставляемом с устанавливаемым программным обеспечением. Такую же информацию выводит и команда `configure --help`.

Комментарий

Система конфигурации и сборки GNU позволяет разработчикам программного обеспечения распространять самостоятельно конфигурируемые программные продукты, способные производить сборку на широком спектре систем. Эта система создает сценарий оболочки configure, который подготавливает дистрибутив программного продукта к сборке и установке на локальную систему. Сценарий configure сканирует локальную систему на наличие факторов, влияющих на распространение программного продукта, и создает соответствующие файлы сборки. После запуска сценария configure можно будет произвести сборку программного продукта с помощью команды make и установить его с помощью команды `make install`.

Сценарий configure определяет, какой из C-компиляторов будет использован (обычно это gcc) и задает набор флагов, передаваемых компилятору. Можно также переопределить значения для переменных окружения `CC` и `CFLAGS`. См. раздел «Примеры».

Примечания

В каждом пакете, использующем утилиту автоконфигурации GNU, предоставляется его собственная копия сценария configure, которую разработчик программного обеспечения создает при помощи GNU-утилиты autoconf (www.gnu.org/software/autoconf). Сведения о доступных ключах приведены в файлах README и INSTALL, предоставляемых вместе с устанавливаемым программным обеспечением.

Сценарии configure являются самодостаточными, нормально работают на широком спектре систем. Какие-либо специальные ресурсы для использования configure не требуются.

Если зависимость от используемой системы не учтена, сценарий configure завершает свою работу с сообщением об ошибке.

Примеры

Простейший способ вызова configure заключается в использовании утилиты cd для превращения в рабочий того каталога, в котором находится устанавливаемый программный продукт, и в запуске следующей команды:

```
$ ./configure
```

Символы ./, стоящие перед именем команды, гарантируют запуск именно того сценария configure, который поставлен вместе с устанавливаемым программным продуктом. Например, чтобы заставить configure создать файлы сборки, передающие компилятору gcc флаги -Wall и -O2, нужно из оболочки bash ввести следующую команду:

```
$ CFLAGS="-Wall -O2" ./configure
```

А если используется оболочка tcsh, нужно ввести следующую команду:

```
tcsh $ env CFLAGS="-Wall -O2" ./configure
```

cp

Копирование файлов.

cp [ключи] файл-источник файл-приемник
cp [ключи] список_файлов_источников каталог-приемник

Утилита cp копирует один или несколько файлов. Она либо создает копию одного файла (первый формат), либо копирует один или несколько файлов в каталог (второй формат). При запуске с ключом -R утилита cp может копировать иерархию каталогов.

Аргументы

Файл-источник — это путевое имя файла, который копируется утилитой cp. ***Файл-приемник*** — это путевое имя, которое утилита cp присваивает создаваемой копии файла.

Список_файлов_источников — это список, состоящий из одного или нескольких путевых имен файлов, которые копируются утилитой cp. ***Каталог-приемник*** — это путевое имя каталога, в который cp помещает копируемые файлы. При использовании этого формата утилита cp дает каждому копируемому файлу такое же простое имя файла, которое было у его файла-источника.

Ключ -R позволяет cp копировать иерархию каталогов в рекурсивном режиме из ***списка_файлов_источников*** в ***каталог-приемник***.

Ключи

При работе под управлением Linux утилита cp допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--archive -a` Осуществление попытки сохранить владельца, группу, права доступа, дату последнего обращения и дату последнего изменения исходного файла (файлов) при копировании в рекурсивном режиме без разыменования символьных ссылок. Аналогично применению ключа -dpR. (Linux)

`--backup -b` Если в результате копирования файла будет удаляться или переписываться существующий файл, этот ключ заставит создать резервную копию того файла, который будет переписан. Резервная копия будет иметь такое же имя, что и у файла-

источника, но с добавлением к нему знака тильды (~). При совместном использовании ключей --backup и --force утилита cp создает резервную копию при попытке скопировать файл в самого себя. Сведения о других ключах для создания резервных копий даны в info-странице, посвященной основным утилитам (core utils). (Linux)

-d Для каждого файла, являющегося символьной ссылкой, копирование символьной ссылки, а не файла, на который она указывает. Кроме этого сохранение жестких ссылок в ***файл-приемнике***, существующих между соответствующими ***файлами-источниками***. Этот ключ является эквивалентом ключей --no-dereference и --preserve=links. (Linux)

--force -f Когда ***файл-приемник*** уже существует и не может быть открыт для записи, этот ключ заставляет утилиту cp попытаться удалить ***файл-приемник*** перед копированием ***файла-источника***. Этот ключ полезен в том случае, когда пользователь, копирующий файл, не имеет права на запись в уже существующий файл, но имеет право на запись в каталог, содержащий ***файл-приемник***. Чтобы создать резервную копию файла-источника перед его перезаписью или удалением, следует воспользоваться ключом -b.

-H (с частичным разыменованием). Для каждого файла, являющегося символьной ссылкой, копирование файла, на который указывает ссылка, а не самой символьной ссылки. Действие этого ключа распространяется на файлы, указанные в командной строке, оно не касается файлов, найденных при спуске по иерархии каталога. При работе под управлением OS X этот ключ работает только с ключом -R. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L»

--interactive -i Вывод приглашения на подтверждение перед каждым намерением cp переписать файл. Если строка ответа будет начинаться с буквы y или буквы Y, cp скопирует файл. Если ввести что-нибудь другое, cp не станет копировать файл.

--dereference -L (с разыменованием). Для каждого файла, являющегося символьной ссылкой, копирование файла, на который указывает ссылка, а не самой символьной ссылки. Действие этого ключа распространяется на все файлы. При работе под управлением OS X этот ключ работает только с ключом -R. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L».

--no-dereference -P (без разыменования). Для каждого файла, являющегося символьной ссылкой, копирование символьной ссылки, а не файла, на который она указывает. Действие этого ключа распространяется на все файлы. При работе под управлением OS X этот ключ работает только с ключом -R. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «chgrp», в подразделе «Примеры».

--preserve [=*attr*] -p Создание файла-приемника с тем же владельцем, группой, правами доступа, датой последнего обращения и датой последнего изменения,

которые были у ***файла-источника***. Ключ -p не допускает использование аргумента. Без аргумента ***attr*** ключ --preserve работает с соответствии с прежним описанием. Аргумент ***attr*** является списком с запятой в качестве разделителя, который может включать в себя поля режима — mode (указание прав доступа и ACL-списки), принадлежности — ownership (указание владельца и группы), отметок времени — timestamps (указание дат последнего обращения и изменения), связей — links (указание жестких ссылок) и всего остального — all (указание всех остальных атрибутов).

--parents Копирование относительного путевого имени в каталог при создании каталогов по мере необходимости. См. раздел «Примеры». (Linux)

--recursive
-R
или -r Рекурсивное копирование иерархий каталогов, включая обычные файлы. При работе под управлением Linux подразумевается применение ключа --no-dereference (-d): с ключами -R, -r или --recursive, утилита cp копирует ссылки (а не файлы, на которые они указывают). Ключи -r и --recursive доступны только при работе под управлением Linux.

--update -u Копирование только в том случае, когда ***файл-приемник*** не существует или когда он по сравнению с ***файлом-источником*** является уже устаревшим (то есть этот ключ не допустит перезаписи более новой версии ***файла-приемника***). (Linux)

--verbose -v Вывод имени каждого файла по мере его копирования утилитой cp.

Примечания

При работе под управлением Linux утилита cp разыменовывает символьные ссылки, пока не будет использован один или несколько ключей из числа -R, -r, --recursive, -P, -d или --no-dereference. В разделе «Ключи» уже давалось объяснение, что при работе под управлением Linux использование ключа -H приводит к разыменованию только тех символьных ссылок, которые перечислены в командной строке. При работе под управлением Mac OS X утилита cp без ключа -R всегда разыменовывает символьные ссылки, а с ключом -R она их не разыменовывает (по умолчанию используется ключ -P), пока не будет указан ключ -H или ключ -L.

При работе под управлением Linux с утилитой cp может использоваться множество ключей, полный список которых приведен на info-странице, посвященной основным утилитам (core utils).

Если до выполнения команды cp ***файл-приемник*** уже существовал, cp переписывает этот файл, уничтожая его содержимое, но оставляя прежние права доступа, владельца и группу, связанную с файлом.

Если ***файла-приемника*** не было, cp использует права доступа, существовавшие к ***файлу-источнику***. Пользователь, копирующий файл, становится владельцем ***файла-приемника***, а группа, в которой зарегистрирован пользователь, становится группой, которая связана с ***файлом-приемником***.

Использование ключа -p без аргументов заставляет утилиту cp предпринять попытку установки владельца, группы, прав доступа, даты последнего обращения и даты последнего изменения в соответствии с аргументами ***файла-источника***.

В отличие от утилиты ln (см. соответствующий раздел), ***файл-приемник***, создаваемый утилитой cp, не зависит от своего ***файла-источника***.

При работе под управлением Mac OS X, начиная с версии 10.4, утилита cp копирует расширенные атрибуты.

Примеры

Первая команда создает копию файла letter в рабочем каталоге. Копии присваивается имя letter.sav.

```
$ cp letter letter.sav
```

Следующая команда копирует все файлы, имена которых оканчиваются на .c, в каталог archives, являющийся подкаталогом рабочего каталога. Каждый скопированный файл сохраняет свое простое имя файла, но приобретает новое абсолютное путевое имя. Ключ -p (--preserve) приводит к тому, что у файлов, копируемых в каталог archives, будут установлены те же владельцы, группы, права доступа, даты последнего обращения и модификации, что и у файлов-источников.

```
$ cp -p *.c archives
```

Следующий пример копирует файл memo из личного каталога пользователя Sam в рабочий каталог:

```
$ cp ~sam/memo .
```

Следующий пример запускается под управлением Linux, и в нем для копирования memo/thursday/max в каталог dir в качестве dir/memo/thursday/max используется ключ --parents. Утилита find показывает только что созданную иерархию каталога.

```
$ cp --parents memo/thursday/max dir
$ find dir
dir
dir/memo
dir/memo/thursday
dir/memo/thursday/max
```

Следующая команда копирует файлы по имени memo и letter в другой каталог. У копий остаются те же простые имена, которые были у исходных файлов (memo и letter), но изменяются абсолютные путевые имена. Абсолютные путевые имена скопированных файлов становятся соответственно /home/sam/memo и /home/sam/letter.

```
$ cp memo letter /home/sam
```

В последней команде показан один из вариантов использования ключа -f (--force). Пользователь Max является владельцем рабочего каталога и безуспешно пытается скопировать один файл поверх другого (me), в отношении которого у него нет прав доступа. Поскольку у него есть право на запись в каталог, в котором находится файл me, Max может удалить файл, но не может провести в него запись. Ключ -f (--force) удаляет ссылки на me или удаляет этот файл, затем копирует файл one в новый файл по имени me.

```
$ ls -ld
drwxrwxr-x    2 max max 4096 Oct 21 22:55 .
$ ls -l
-rw-r--r--    1 root root 3555 Oct 21 22:54 me
-rw-rw-r--    1 max max 1222 Oct 21 22:55 one
```

```
$ cp one me
cp: cannot create regular file 'me': Permission denied
            cp: создание обычного файла 'me' невозможно: Отсутствие прав доступа
$ cp -f one me
$ ls -l
-rw-r--r--    1 max max 1222 Oct 21 22:58 me
-rw-rw-r--    1 max max 1222 Oct 21 22:55 one
```

Если бы в дополнение к ключу `-f` (`--force`) пользователь Max воспользовался ключом `-b` (`--backup`), утилита cp создала бы резервную копию файла me с именем me~. Дополнительные сведения даны в главе 4, в разделе «Права доступа к каталогам» на с. 119.

cpio

Создание архива, восстановление файлов из архива или копирование иерархии каталога.

*cpio — –create|–o **[ключи]***
*cpio — –extract|–i **[ключи] [список_шаблонов]***
*cpio — –pass-through|–p **[ключи] каталог-приемник***

Утилита cpio имеет три режима работы: режим создания — *create* (copy-out), помещающий несколько файлов в файл архива, режим извлечения — *extract* (copy-in), восстанавливающий файлы из архива, и режим прохода — *pass-through* (copy-pass), копирующий иерархию каталога. Создаваемый утилитой cpio архивный файл может быть сохранен на диске, магнитной ленте или на других сменных носителях или на удаленной системе.

В режиме создания (create) со стандартного ввода считывается список имен обычных файлов или файлов каталога и запись итогового архивного файл в стандартный вывод. Этот режим можно использовать для создания архива. В режиме извлечения (extract) архив считывается со стандартного ввода, и происходит извлечение файлов из этого архива. При этом из архива можно восстановить либо все файлы, либо только те из них, чьи имена соответствуют заданному шаблону. В режиме прохода (pass-through) со стандартного ввода считывается список имен обычных файлов или файлов каталога, и эти файлы копируются в указанный каталог.

Аргументы

По умолчанию утилита cpio в режиме извлечения (extract) извлекает все найденные в архиве файлы. Можно задать выборочное извлечение файлов, предоставив утилите ***список_шаблонов***. Если имя файла в архиве соответствует одному из шаблонов из ***списка_шаблонов***, утилита cpio извлекает этот файл, в противном случае он просто игнорируется. Шаблоны в используемом cpio ***списке_шаблонов*** аналогичны групповым символам оболочки (см. главу 5, раздел «Генерация имени файла и расширение имени» на с. 157), за исключением того, что в данном ***списоке_шаблонов*** принимается соответствие в именах файлов слэша (/) и ведущей точки (.).

В режиме прохода (pass-through) утилите cpio нужно обязательно указать в качестве аргумента имя ***каталога-приемника***.

Ключи

Главный ключ указывает режим работы утилиты cpio: создания (create), извлечения (extract) или прохода (pass-through).

Главные ключи

В командную строку нужно включить только один из этих ключей. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--extract -i` (режим извлечения — copy-in). Чтение архива со стандартного ввода и извлечение файлов. В отсутствие в командной строке ***списка_шаблонов***, cpio извлекает из архива все файлы. При использовании ***списка_шаблонов*** cpio извлекает только те файлы, чьи имена соответствуют одному из шаблонов, имеющихся в ***списке_шаблонов***. В следующем примере из устройства, примонтированного к каталогу /dev/sde1, извлекаются только те файлы, чьи имена оканчиваются на .c:

```
$ cpio -i \*.c < /dev/sde1
```

Символ обратного слэша не дает оболочке расширить символ * перед тем, как она передаст аргумент утилите cpio.

`--create -o` (режим создания — copy-out). Создание архива из файлов, указанных в стандартном вводе. Эти файлы могут быть обычными файлами или файлами каталогов, и имя каждого из них должно быть представлено в отдельной строке. Архив по мере своего создания записывается на стандартный вывод. Часто для генерации имен файлов, используемых утилитой cpio, применяется утилита find. Следующая команда создает архив всей локальной системы и записывает его в устройство, примонтированное к каталогу /dev/sde1:

```
$ find / -depth -print | cpio -o >/dev/sde1
```

Ключ -depth заставляет утилиту find искать файлы, начиная с тех, которые глубже всех находятся в иерархии, снижая тем самым вероятность проблем с правами доступа при восстановлении этих файлов из архива. См. комментарий, посвященный этому ключу в разделе «Комментарии».

`--pass-through -p` (режим прохода — copy-pass). Копирует файлы из одного места в системе в другое. Вместо создания файла архива, содержащего файлы, указанные в стандартном вводе, утилита cpio копирует их в ***каталог-приемник*** (последний аргумент в командной строке cpio). Эффект получается таким же, как при создании архива в режиме создания, а затем извлечения из него файлов в режиме извлечения, за исключением того, что режим прохода не создает архив. В следующем примере файлы из рабочего каталога и всех его подкаталогов копируются в каталог ~max/code:

```
$ find . -depth -print | cpio -pdm ~max/code
```

Другие ключи

Следующие ключи позволяют изменять поведение утилиты cpio. Эти ключи работают с одним или несколькими предыдущими главными ключами.

Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--reset-access-time` `-a` — Восстановление времени последнего обращения к исходным файлам после их копирования, чтобы у них после копирования было такое время последнего обращения, как и до него.

`-B` — (block — размер блока). Установка размера блока в 5120 байт вместо используемого по умолчанию размера в 512 байт. При работе под управлением Linux этот ключ влияет на размер блоков на вводе и выводе, а при работе под управлением OS X он влияет только на размер блоков на выводе.

`--block-size=`*n* — Установка размера блока для ввода и вывода, равным *n* блоков по 512 байт. (Linux)

`-c` — (compatible — совместимый режим). Запись заголовочной информации в коде ASCII, чтобы устаревшие (несовместимые с новыми) версии утилиты cpio на других системах могли читать файл. Потребности в использовании данного ключа возникают крайне редко.

`--make-directories` `-d` — Создание каталогов при копировании файлов по мере надобности. Например, потребность в этом ключе возникает при извлечении файлов из архива, при создании которого был использован список файлов, созданный утилитой find с ключом `-depth`. Этот ключ может быть использован только с ключами `-i` (`--extract`) и `-p` (`--pass-through`).

`--pattern-file=` ***имя_файла*** `-E` ***имя_файла*** — Построчное чтение ***списка_шаблонов*** из файла с именем ***имя_файла***. Кроме этого ***список_шаблонов*** можно указать в командной строке.

`--file=`***архив*** `-F` ***архив*** — Использование ***архива*** в качестве имени файла архива. В режиме извлечения — чтение из ***архива***, а не со стандартного ввода. В режиме создания — запись в ***архив***, а не в стандартный вывод. Этот ключ может использоваться для обращения по сети к устройству на другой системе. Дополнительные сведения даны в описании ключа `-f` (`--file`) в разделе, посвященном утилите tar.

`--nonmatching` `-f` — (flip — переворот). Инвертирование смысла теста, проводимого со ***списком_шаблонов*** при извлечении файлов из архивов. Файлы извлекаются из архива, только если их имя не соответствует ни одному из шаблонов из ***списка_шаблонов***.

`--help` — Вывод списка ключей. (Linux)

`--dereference` `-L` — Для каждого файла, являющегося символьной ссылкой, копирование файла, на который указывает ссылка (а не самой символьной ссылки). При использовании этого ключа файлы, не являющиеся символьными ссылками, обрабатываются обычным образом.

`--link -l` — По возможности, создание ссылок на файлы вместо их копирования.

`--preserve-modification-time` `-m` Сохранение времени последнего изменения файлов, извлекаемых из архива. Без этого ключа файлы показывают время, в которое они были извлечены. При использовании этого ключа созданные файлы показывают время, которое у них было на момент копирования их в архив.

`--no-absolute-filenames` В режиме извлечения создание всех имен файлов относительно рабочего каталога, даже для тех файлов, которые были заархивированы с абсолютными путевыми именами. (Linux)

`--rename -r` Разрешение переименования файлов по мере копирования их утилитой cpio. Когда cpio выводит приглашение на ввод имени файла, в ответ вводится новое имя. Затем файл копируется под новым именем. Если вместо ввода имени нажать ВВОД, cpio этот файл не копирует.

`--list -t` (table of contents — содержимое). Вывод содержимого архива. Этот ключ работает только с ключом `-i` (`--extract`), хотя на самом деле файлы из архива не извлекаются. С ключом `-v` (`--verbose`) этот ключ заставляет утилиту выводить подробное описание содержимого в формате, аналогичном используемому командой `ls -l`.

`--unconditional` `-u` Переписывание существующих файлов, независимо от времени их последнего изменения. Без этого ключа утилита cpio не станет переписывать более новые версии файлов их устаревшими версиями и выведет предупреждающее сообщение.

`--verbose -v` Вывод списка файлов по мере их обработки. С ключом `-t` (`--list`) заставляет утилиту выводить подробное описание содержимого в формате, аналогичном используемому командой `ls -l`.

Комментарии

Без ключа `-u` (`--unconditional`) утилита cpio не будет переписывать более новые версии файла его устаревшими версиями.

При создании архива можно использовать в качестве ввода как имена обычных файлов, так и имена каталогов. Если имя обычного файла появится в списке ввода до имени своего родительского каталога, то обычный файл также появится в архиве до своего родительского каталога. Такой порядок может привести к ошибке, которую можно было бы избежать: при извлечении файлов из архива дочернему файлу негде будет появляться в файловой структуре, поскольку его родительский каталог еще не извлечен.

Решить эту проблему, обеспечив появление файла в архиве после его родительского каталога, удается не всегда. Одна из проблем возникает, когда файлы извлекаются с ключом `-m` (`--preserve-modification-time`). Поскольку время последнего изменения родительского каталога обновляется, как только внутри него создается файл, исходное время изменения родительского каталога утрачивается, как только в него будет записан первый из файлов.

Решение этой потенциальной проблемы заключается в обеспечении того, чтобы все файлы при создании архива появлялись раньше своих родительских каталогов, и в создании каталогов при извлечении файлов из архива по мере надобности. При использовании такой технологии каталоги извлекаются только после того, как в них будут записаны все файлы, и время последнего изменения сохраняется.

Работая с ключом -depth, утилита find создает список файлов, в котором все дочерние файлы появляются в списке перед своими родительскими каталогами. Если этот список используется для создания архива, то файлы располагаются в нужном для этого порядке. (См. первый пример в следующем разделе.) При извлечении файлов из архива ключ -d (--make-directories) заставляет утилиту cpio по мере необходимости создавать родительские каталоги, а ключ -m (--preserve-modification-time), в соответствии со своим названием, заставляет сохранять время последнего изменения. Использование этой комбинации утилит и ключей приводит к сохранению времени последнего изменения каталогов в последовательности действий по созданию архива и извлечению из него файлов.

Такая стратегия к тому же решает и другие потенциальные проблемы. Иногда для родительского каталога могут отсутствовать права на извлечение в него файлов. Когда утилита cpio автоматически создает каталог, работая с ключом -d (--make-directories), насчет наличия прав записи в каталог можно быть спокойным. Когда каталог извлекается из архива (после того как все файлы записаны в каталог), он извлекается со своими исходными правами доступа к нему.

Примеры

Первый пример создает архив файлов, имеющихся в личном каталоге пользователя Sam, записывая архив на USB-накопитель, примонтированный к каталогу /dev/sde1.

```
$ find /home/sam -depth -print | cpio -oB >/dev/sde1
```

Утилита find выдает имена файлов, которые утилита cpio использует для создания архива. Ключ -depth заставляет все записи в каталоге попадать в список перед помещением в него имени самого каталога, давая возможность cpio сохранять каталогам исходное время их последнего изменения (см. предыдущий раздел «Комментарии»). При извлечении файлов из архива (см. следующие примеры) следует использовать ключи -d (--make-directories) и -m (--preserve-modification-time). Ключ -B устанавливает размер блока равным 5120 байт.

При работе под управлением Mac OS X, личный каталог хранится не в каталоге /home, а в каталоге /Users.

Для проверки содержимого файла архива и вывода подробного списка содержащихся в нем файлов нужно ввести следующую команду:

```
$ cpio -itv < /dev/sde1
```

Следующая команда восстанавливает файлы, которые раньше были в подкаталоге memo личного каталога пользователя Sam:

```
$ cpio -idm /home/sam/memo/\* < /dev/sde1
```

Ключ -d (--make-directories) обеспечивает воссоздание по мере надобности любого подкаталога, который присутствовал в каталоге memo. Ключ -m (--preserve-modification-time) заставляет сохранить время последнего изменения файлов и каталогов. Звездочка в регулярном выражении предварена символом отключения, чтобы оболочка не занялась ее расширением.

Следующая команда аналогична предыдущей, за исключением использования Linux-ключа --no-absolute-filenames для воссоздания каталога memo в рабочем каталоге по имени memocopy. Шаблон не начинается со слэша, представляющего корневой каталог, позволяя утилите cpio создавать файлы с относительными путевыми именами.

```
$ pwd
/home/sam/memocopy
$ cpio -idm --no-absolute-filenames home/sam/memo/\* < /dev/sde1
```

В последнем примере используется ключ `-f`, заставляющий восстановить все файлы, имеющиеся в архиве, за исключением тех, которые ранее были в подкаталоге memo:

```
$ cpio -ivmdf /home/sam/memo/\* < /dev/sde1
```

Ключ `-v` заставляет вывести список извлеченных файлов по мере обработки архива утилитой cpio, позволяя проверить извлечение ожидаемых файлов.

crontab

Обслуживание crontab-файлов.

*crontab [–u **имя_пользователя**] **имя_файла***
*crontab [–u **имя_пользователя**] **ключ***

Crontab-файл связывает периодические моменты времени (например, 14:00 каждую среду) с командами. Утилита cron выполняет каждую команду в указанное время. Когда вы работаете под своим собственным именем пользователя, утилита crontab устанавливает, удаляет, выводит списком и позволяет редактировать ваши crontab-файлы. Пользователь, работающий с root-правами, может работать с crontab-файлами любого пользователя.

Аргументы

Первый формат используется для копирования содержимого файла с именем ***имя_файла*** (который содержит crontab-команды) в crontab-файл того пользователя, который запустил команду, или того пользователя, чье имя указано в ***имени_пользователя***. Когда у пользователя нет crontab-файла, этот процесс создает новый файл, а когда у него есть crontab-файл, этот процесс его переписывает. Если вместо ***имени_файла*** ставится дефис (`-`), crontab читает команды со стандартного ввода.

Второй формат используется в зависимости от указанного ключа для вывода списка команд, удаления или редактирования crontab-файла.

Ключи

Можно выбрать только один из ключей: `-e`, `-l` или `-r`. Пользователь, работающий с root-правами, может использовать с одним из этих ключей ключ `-u`.

`-e` (edit — редактирование). Запуск текстового редактора, указанного переменной оболочки `VISUAL` или `EDITOR` для редактирования crontab-файла, что дает возможность добавлять, изменять или удалять записи. Этот ключ используется для установки измененного crontab-файла при выходе из редактора.

`-l` (list — вывод списка). Вывод содержимого crontab-файла.

`-r` (remove — удаление). Удаление crontab-файла.

`-u` ***имя_пользователя*** (user — указание пользователя). Работа с crontab-файлом, принадлежащим указанному пользователю. Этот ключ может использовать только пользователь, работающий с root-правами.

Примечания

В этом разделе рассматриваются версии утилит cron, crontab и crontab-файлов, которые были созданы Полем Викси (Paul Vixie), — поэтому эта версия утилиты cron называется Vixie cron. Эта версия является POSIX-совместимой и отличается от более ранней версии Vixie cron, а также имеет синтаксис, отличный от классического синтаксиса SVR3.

Пользовательские crontab-файлы хранятся в каталоге /var/spool/cron или в каталоге /var/spool/cron/crontabs. Каждый файл носит имя пользователя, которому он принадлежит.

Системная утилита по имени cron читает crontab-файлы и запускает команды. Если командная строка в crontab-файле не перенаправляет вывод этой утилиты, то весь вывод, отправляемый на стандартный вывод и стандартную ошибку, отправляется пользователю по электронной почте, если только переменная MAILTO внутри crontab-файла не указывает на другое имя пользователя.

Для упрощения работы системного администратора каталоги, называемые ежечасными — /etc/cron.hourly, ежедневными — /etc/cron.daily, еженедельными /etc/cron.weekly и ежемесячными — /etc/cron.monthly, хранят crontab-файлы, которые на большинстве систем запускаются с помощью утилиты run-parts, которая, в свою очередь, запускается с помощью файла /etc/crontab.

Каждый из этих каталогов содержит файлы, выполняющие системные задачи с интервалами, соответствующими имени каталога. Пользователь, работающий с root-правами, может добавлять файлы к этим каталогам вместо добавления строк к корневым crontab-файлам. Типовой файл /etc/crontab имеет следующий вид:

```
$ cat /etc/crontab
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/

# run-parts
01 * * * * root run-parts /etc/cron.hourly
02 4 * * * root run-parts /etc/cron.daily
22 4 * * 0 root run-parts /etc/cron.weekly
42 4 1 * * root run-parts /etc/cron.monthly
```

Каждая запись в crontab-файле начинается с пяти полей, в которых указывается, когда команда должна быть запущена (минута, час, день месяца, месяц и день недели). Утилита cron интерпретирует звездочку, которая ставится вместо числа как групповой символ, представляющий любые возможные значения. В поле дня недели для представления воскресенья можно использовать число 7 или число 0.

Лучше запускать задания для утилиты cron не в ровные отметки времени, совпадающие с часом, получасом или четвертью часа, а сдвигать их запуск на произвольное количество минут до или после такой отметки. Когда задания запускаются в произвольные отметки времени, это снижает вероятность одновременного запуска множества процессов, а следовательно, и вероятность перегрузки системы.

При запуске cron (обычно это происходит в процессе загрузки системы) эта утилита считывает все crontab-файлы в память. Основную часть времени утилита cron

бездействует, активизируясь раз в минуту, проверяя все crontab-записи, сохраненные ею в памяти, и запуская задания, чей срок запуска уже подошел.

cron.allow, cron.deny — Перечень пользователей, имеющих право запускать утилиту crontab, определяется пользователем, работающим с root-правами путем создания, редактирования и удаления разрешающего файла cron.allow и запрещающего файла cron.deny. При работе под управлением Linux эти файлы хранятся в каталоге /etc, а при работе под управлением Mac OS X они хранятся в каталоге /var/at (на который имеется символьная ссылка /usr/lib/cron). При создании файла cron.deny без записей и в отсутствие файла cron.allow пользоваться утилитой crontab может любой пользователь. Если существует файл cron.allow, пользоваться crontab могут лишь те пользователи, которые перечислены в этом файле, независимо от присутствия и содержимого файла cron.deny. В противном случае можно перечислить в файле cron.allow имена тех пользователей, кому разрешено использование утилиты crontab, а в файле cron.deny перечислить имена тех пользователей, которым пользоваться этой утилитой запрещено. (В перечислении пользователей в файле cron.deny нет особой нужды, поскольку если существует файл cron.allow и пользователь в нем отсутствует, то ему все равно не удастся воспользоваться утилитой crontab.)

Примеры

В следующих примерах для вывода содержимого crontab-файла пользователя Sam (/var/spool/cron/sam) используется команда `crontab -l`. Все сценарии, запускаемые пользователем Sam, находятся в его каталоге ~/bin. В первой строке переменной `MAILTO` присваивается значение `max`, чтобы не перенаправленный вывод из команд, запущенных из crontab-файла пользователя Sam, получал пользователь Max. Сценарий sat.job запускается каждую субботу (день 6) в 2:05 ночи (AM), сценарий twice.week запускается в 12:02 ночи (AM) в воскресенье и четверг (дни 0 и 4), а сценарий twice.day запускается дважды в день ежедневно 10:05 утра (AM) и 4:05 после полудня (PM).

```
$ who am i
sam

$ crontab -l
MAILTO=max
05 02 * * 6 $HOME/bin/sat.job
02 00 * * 0,4 $HOME/bin/twice.week
05 10,16 * * * $HOME/bin/twice.day
```

Чтобы добавить запись к своему crontab-файлу, нужно запустить утилиту crontab с ключом -e (edit). На некоторых Linux-системах используется версия crontab, не поддерживающая ключ -e. Если на локальной системе запускается такая версия, нужно сделать копию вашего существующего crontab-файла, отредактировать ее, а затем передать ее утилите crontab еще раз, как показано в следующем примере. Ключ -l (list) заставляет вывести копию вашего crontab-файла.

```
$ crontab -l > newcron
$ vim newcron
...
$ crontab newcron
```

cut

Выбор символов или полей из строк ввода.

cut [ключи] [список_файлов]

Утилита cut выбирает символы или поля из строк ввода и записывает их в стандартный вывод.

Нумерация символов и полей начинается с единицы.

Аргументы

Список_файлов — это список обычных файлов. Если аргумент не указан или если вместо него указан дефис (-), cut считывает строки со стандартного ввода.

Ключи

При работе под управлением Linux cut допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--characters=`*clist*
`-c` *clist*
: Выбор символов, заданных номерами столбцов из списка ***clist***. Значением ***clist*** является один или несколько номеров столбцов, разделенных запятыми, или диапазон столбцов. Диапазон указывается двумя номерами столбцов, разделенными дефисом. Диапазон *-n* означает столбцы с первого по *n*-й, а диапазон *n-* означает столбцы с *n*-го и до конца строки.

`--delimiter=`*dchar*
`-d` *dchar*
: Указание символа ***dchar*** в качестве разделителя полей. Также указание ***dchar*** в качестве разделителя полей вывода, если только не будет использован ключ `--output-delimiter`. По умолчанию разделителем служит символ табуляции. Чтобы защитить символы от расширения оболочкой, их при необходимости нужно процитировать.

`--fields=`*flist*
`-f` *flist*
: Выбор полей, указанных в списке ***flist***. Значением ***flist*** является один или несколько номеров полей или диапазон полей. Диапазон указывается двумя номерами полей, разделенными дефисом. Диапазон *-n* означает поля с первого по *n*-й, а диапазон *n-* означает поля с *n*-го и до последнего. Разделителем полей служит символ табуляции, если только для изменения разделителя не будет использован ключ `--delimiter`.

`--output-delimiter=`*ochar*
: Указание символа ***ochar*** в качестве разделителя полей вывода. По умолчанию разделителем служит символ табуляции. Другой разделитель можно указать с помощью ключа `--delimiter`. Чтобы защитить символы от расширения оболочкой, их при необходимости нужно процитировать.

`--only-delimited -s`
: Копирование только строк, содержащих разделители. Без этого ключа утилита cut копирует, но не изменяет строки, не содержащие разделителей.

Примечания

Несмотря на ограниченные возможности, утилита cut легка в усвоении и использовании и представляет собой хороший вариант, когда столбцы и поля могут быть выбраны без использования соответствия шаблонам. Иногда утилита cut используется вместе с утилитой paste (см. соответствующий раздел).

Примеры

Для следующих двух примеров нужно предположить, что команда `ls -l` имеет следующий вывод:

```
$ ls -l
total 2944
-rwxr-xr-x  1 zach pubs     259 Feb  1 00:12 countout
-rw-rw-r--  1 zach pubs    9453 Feb  4 23:17 headers
-rw-rw-r--  1 zach pubs 1474828 Jan 14 14:15 memo
-rw-rw-r--  1 zach pubs 1474828 Jan 14 14:33 memos_save
-rw-rw-r--  1 zach pubs    7134 Feb  4 23:18 tmp1
-rw-rw-r--  1 zach pubs    4770 Feb  4 23:26 tmp2
-rw-rw-r--  1 zach pubs   13580 Nov  7 08:01 typescript
```

Следующая команда выводит права доступа к файлам в рабочем каталоге. Утилита cut с ключом `-c` выбирает из каждой строки ввода символы со второго по десятый. Символы из этого диапазона записываются в стандартный вывод.

```
$ ls -l | cut -c2-10
otal 2944
rwxr-xr-x
rw-rw-r--
rw-rw-r--
rw-rw-r--
rw-rw-r--
rw-rw-r--
rw-rw-r--
```

Следующая команда выводит размер и имя каждого файла в рабочем каталоге. Ключ `-f` заставляет выбирать из строк ввода пятые и десятые поля. Ключ `-d` заставляет утилиту cut использовать в качестве разделителей не символы табуляции, а пробелы. Утилита tr (см. соответствующий раздел) с ключом `-s` заменяет череду из нескольких пробелов одним пробелом, в противном случае утилита cut посчитает лишние пробелы в качестве отдельных полей.

```
$ ls -l | tr -s '  ' ' ' | cut -f5,9 -d' '
259 countout
9453 headers
1474828 memo
1474828 memos_save
7134 tmp1
4770 tmp2
13580 typescript
```

В последнем примере выводится список полных имен в том виде, в котором они сохранены в пятых полях файла /etc/passwd. Ключ `-d` указывает, что в качестве раз-

делителя поля будет использоваться двоеточие. Хотя этот пример работает и под управлением Mac OS X, каталог /etc/passwd не содержит информацию о большинстве пользователей, за ней нужно обратиться к каталогу Open Directory.

```
$ cat /etc/passwd
root:x:0:0:Root:/:/bin/sh
sam:x:401:50:Sam the Great:/home/sam:/bin/zsh
max:x:402:50:Max Wild:/home/max:/bin/bash
zach:x:504:500:Zach Brill:/home/zach:/bin/tcsh
hls:x:505:500:Helen Simpson:/home/hls:/bin/bash

$ cut -d: -f5 /etc/passwd
Root
Sam the Great
Max Wild
Zach Brill
Helen Simpson
```

date

Вывод или установка системного времени и даты.

date [ключи] [+формат]
date [ключи] [новая_дата]

Утилита date выводит известные системе время и дату. Пользователь, работающий с root-правами, может использовать date для изменения показаний системных часов.

Аргументы

Аргумент ***+формат*** указывает формат для вывода даты. Строка ***формат***, состоящая из полей описателей и текста, следует за знаком плюс (+). Описатели полей предваряются знаком процента, и каждые данные заменяются в выводе их значением. Некоторые описатели полей перечислены в табл. V.10.

Таблица V.10. Избранные описатели полей

Описатель	Значение
%a	Сокращенное название дня недели — от Sun до Sat (от воскресенья до субботы)
%A	Полное название дня недели — от Sunday до Saturday (от воскресенья до субботы)
%b	Сокращенное название месяца — от Jan до Dec (от января до декабря)
%B	Полное название месяца — от January до December (от января до декабря)
%c	Дата и время в формате, используемом для даты по умолчанию
%d	День месяца — от 01 до 31
%D	Дата в формате мм/дд/гг
%H	Час — от 00 до 23
%I	Час — от 00 до 12
%j	Дата по юлианскому календарю (день года — от 001 до 366)
%m	Месяц года — от 01 до 12

продолжение ⇩

Таблица V.10 (*продолжение*)

Описатель	Значение
%M	Минуты — от 00 до 59
%n	Разделитель строк
%P	AM или PM (до или после полудня)
%r	Время в системе записи AM/PM
%s	Количество секунд, прошедших с 1 января 1970 года
%S	Секунды — от 00 до 60 (60 подогнанных скоординированных секунд)
%t	Символ табуляции
%T	Время в формате ЧЧ:ММ:СС
%w	День недели — от 0 до 6 (0 = Sunday [воскресенье])
%y	Последние две цифры года — от 00 до 99
%Y	Год в формате четырех цифр (например, 2009)
%Z	Часовой пояс (например, PDT)

По умолчанию утилита **date** заполняет числовые поля нулями. Знак подчеркивания (_), поставленный в поле сразу за знаком процента (%), заставляет утилиту **date** заполнять поля пробельными символами. Дефис (-), поставленный сразу за знаком процента, заставляет утилиту **date** не заполнять поля, то есть выравнивать поле по левому краю.

Утилита **date** предполагает, что в строке формата любой символ, следующий за знаком процента и не являющийся знаком процента, знаком подчеркивания или дефисом или не являющийся описателем поля, является обычным текстом, который она копирует в стандартный вывод. Обычный текст можно использовать для добавления к дате знаков пунктуации и для добавления надписей (например, можно перед датой поместить слово `ДАТА:`). Аргумент ***формат***, если он содержит пробелы или другие символы, имеющие для оболочки специальное значение, следует поместить в одинарные кавычки.

Установка системных часов

Когда пользователь, работающий с root-правами, указывает ***новую_дату***, система изменяет системное время, заставляя его соответствовать новой дате. Аргумент ***новая_дата*** имеет следующий формат:

nnddhhmm[[cc]yy][.ss]

где *nn* — это номер месяца (01–12), *dd* — день месяца (01–31), *hh* — час по 24-часовой шкале (00–23) и *mm* — минуты (00–59).

При изменении системной даты нужно указать как минимум эти поля. Дополнительное поле *cc* указывает первые две цифры года (значение столетия за минусом 1), а *yy* указывает последние две цифры года. За *mm* можно указать *yy* или *ccyy*. Если год не будет указан, утилита **date** предположит, что год не изменился.

С помощью поля *.ss* можно указать количество секунд, прошедших с начала минуты.

Ключи

При работе под управлением Linux **date** допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис

(--), работают только под управлением Linux. За исключением особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--date=` *строка_даты*
`-d` *строка_даты*
: Вывод даты, указанной не текущей датой, а ***строкой_даты***. Это ключ не изменяет показаний системного времени. (Linux)

`--reference=` *файл*
`-r` *файл*
: Вывод времени даты и времени последнего изменения ***файла***. (Linux)

`--utc` *или* `--universal` `-u`
: Вывод или установка времени и даты с использованием всеобщего скоординированного времени — Universal Coordinated Time (UTC). По-другому UTC называется временем по Гринвичу — Greenwich Mean Time (GMT).

Примечания

Если установлена база данных локальных настроек, утилита date использует эту базу данных для подстановки элементов, соответствующих локальным настройкам.

Примеры

В первом примере показывается, как установить дату на 2:07:30 PM on August 19 (на 14:07:30 19 августа), не меняя года:

```
# date 08191407.30
Fri Aug 19 14:07:30 PDT 2009
```

В следующем примере показывается аргумент ***формат***, заставляющий утилиту date вывести дату в широко применяемом формате:

```
$ date '+Today is %h %d, %Y'
Today is Aug 19, 2009
```

dd

Преобразование и копирование файла.

*dd [**аргументы**]*

Утилита dd (копирование device-to-device — с устройства на устройство) осуществляет преобразование и копирование файла. Основным предназначением утилиты dd является копирование файлов как на жесткий диск и сменные носители, так и копирование файлов с этих устройств. Она может работать с разделами жесткого диска и создавать точные поблочные образы дисков. Зачастую dd справляется с переносом информации на другие операционные системы и из них, в то время как другие методы терпят неудачу. Богатый набор аргументов позволяет достичь тонкого управления характеристиками переноса данных.

Аргументы

При работе под управлением Linux утилита dd допускает использование общих ключей, описание которых дано в табл. V.2. По умолчанию dd копирует стандартный ввод на стандартный вывод.

`bs=`*n* (block size — размер блока). Чтение и запись блоками по *n* байт. Этот аргумент отменяет действие аргументов `ibs` и `obs`.

`cbs=`*n* (conversion block size — размер преобразуемого блока). Преобразование блоками по *n* байт при осуществлении преобразования в процессе копирования.

`conv=`*тип* `[,`*тип*`...]` Преобразование копируемых данных в порядке, заданном в командной строке путем указания ***типов*** преобразования, которые должны быть разделены запятыми без пробелов. Типы преобразования показаны в табл. V.11.

`count=`***количество_блоков*** Ограничение количества блоков ввода, копируемого утилитой dd, указанным ***количеством_блоков***. Размер каждого блока — это число байт, указанное с помощью аргумента `bs` или аргумента `ibs`.

`ibs=`*n* (input block size — размер блока ввода). Чтение блоками по *n* байт.

`if=`***имя_файла*** (input file — файл ввода). Чтение не со стандартного ввода, а из файла по имени ***имя_файла***. Для чтения с какого-нибудь устройства в качестве ***имени_файла*** можно использовать имя этого устройства.

`obs=`*n* (output block size — размер блока вывода). Запись блоками по *n* байт.

`of=`***имя_файла*** (output file — файл вывода). Запись не в стандартный вывод, а в файл по имени ***имя_файла***. Для записи на какое-нибудь устройство в качестве ***имени_файла*** можно использовать имя устройства.

`seek=`***количество_блоков*** Перед записью любого вывода пропуск указанного ***количества_блоков***. Размер каждого блока — это число байт, указанное с помощью аргумента `bs` или аргумента `obs`.

`skip=`***количество_блоков*** Перед началом копирования пропуск указанного ***количества_блоков*** ввода. Размер каждого блока — это число байт, указанное с помощью аргумента `bs` или аргумента `ibs`.

Таблица V.11. Типы преобразования

тип	Значение
`ascii`	Преобразование символов в кодировке EBCDIC в ASCII, позволяющее читать магнитные ленты, записанные на универсальных машинах IBM и на аналогичных им компьютерах
`block`	При каждом считывании строки ввода (то есть последовательности символов, заканчивающейся разделителем строк) вывод блока текста без разделителя строк. Каждый блок вывода имеет размер, заданный аргументом `bs` или аргументом `obs` и созданный с помощью дополнения текста замыкающими пробелами до заданного размера
`ebcdic`	Преобразование символов в кодировке ASCII в EBCDIC, позволяющее записывать магнитные ленты для использования на универсальных машинах IBM и на аналогичных им компьютерах

тип	Значение
lcase	Преобразование в процессе копирования букв верхнего регистра в буквы нижнего регистра
noerror	Обычно при возникновении ошибки чтения утилита dd завершает работу. Это преобразование позволяет dd продолжить обработку данных, и применяется при попытке восстановления данных с плохого носителя
notrunc	Отмена усечения файла вывода перед началом записи в него
ucase	Преобразование в процессе копирования букв нижнего регистра в буквы верхнего регистра
unblock	Выполнение обратного блочного преобразования

Примечания

При работе под управлением Linux для упрощения указания больших размеров блоков можно использовать стандартные мультипликативные суффиксы, показанные в табл. V.1. При работе под управлением Mac OS X можно также применять некоторые стандартные мультипликативные суффиксы, но вместо показанных в этой таблице букв верхнего регистра в OS X используются строчные буквы, кроме того, при работе под управлением OS X утилита dd поддерживает суффиксы b (блок, умножение на 512) и w (слово, умножение на количество байтов, используемое для представления целого числа).

Примеры

Утилита dd может использоваться для создания файла, заполненного псевдослучайными байтами.

```
$ dd if=/dev/urandom of=randfile2 bs=1 count=100
```

Предыдущая команда производит чтение из файла /dev/urandom (который служит интерфейсом к имеющемуся в ядре генератору случайных чисел) и ведет запись в файл по имени randfile. Размер блока равен 1, а счетчик count равен 100, поэтому randfile имеет длину 100 байт. Чтобы получить более близкие к случайным числа, нужно считывать их из устройства /dev/random. Дополнительные сведения даны на man-страницах urandom и random. При работе под управлением OS X устройства urandom и random ведут себя одинаково.

Утилиту dd можно также использовать для создания точной копии раздела диска. Но при этом нужно проявлять осмотрительность — следующая команда полностью уничтожает все содержимое раздела /dev/sdb1.

```
# dd if=/dev/sda1 of=/dev/sdb1
```

Полное уничтожение файла

Такая же технология может быть использована для полного уничтожения данных из файла перед его удалением, делая практически невозможным восстановление данных из удаленного файла. Потребности в полном уничтожении файла могут возникать из соображений безопасности, для повышения степени которой файл нужно уничтожить несколько раз.

В следующем примере утилита ls показывает размер файла по имени secret, а затем утилита dd с размером блока равным 1 и счетчиком count, соответствующим количе-

ству байт в файле secret, уничтожает файл. Аргумент conv=notrunc обеспечивает запись данных утилитой dd поверх данных, содержащихся в файле, а не в каком-нибудь другом месте диска.

```
$ ls -l secret
-rw-rw-r--  1 max max 2494 Feb  6 00:56 secret
$ dd if=/dev/urandom of=secret bs=1 count=2494 conv=notrunc
2494+0 records in
2494+0 records out
$ rm secret
```

Копирование дискеты

Утилиту dd можно использовать для точного копирования дискеты. Сначала нужно скопировать содержимое дискеты в файл на жестком диске, а затем скопировать файл из жесткого диска на отформатированную дискету. Эта технология работает независимо от содержимого дискеты. В следующем примере копируется дискета, отформатированная в DOS. (Имя файла для гибкого диска на локальной системе может отличаться от имени, приведенного в примере.) Последовательность команд mount, ls, umount в начале и в конце примера позволяет проверить, что на оригинальной дискете и на ее копии содержатся одни и те же файлы. Для создания нескольких копий дискеты можно воспользоваться файлом floppy.copy.

```
# mount -t msdos /dev/fd0H1440 /mnt
# ls /mnt
abprint.dat  bti.ini      setup.ins    supfiles.z   wbt.z
adbook.z     setup.exe    setup.pkg    telephon.z
# umount /mnt
# dd if=/dev/fd0 ibs=512 > floppy.copy
2880+0 records in
2880+0 records out
# ls -l floppy.copy
-rw-rw-r--   1 max speedy 1474560 Oct 11 05:43 floppy.copy
# dd if=floppy.copy bs=512 of=/dev/fd0
2880+0 records in
2880+0 records out
# mount -t msdos /dev/fd0H1440 /mnt
# ls /mnt
abprint.dat  bti.ini      setup.ins    supfiles.z   wbt.z
adbook.z     setup.exe    setup.pkg    telephon.z
# umount /mnt
```

df

Вывод информации об использовании дискового пространства.

df [ключи] [список_файловых_систем]

Утилита df (disk free — место на диске) сообщает о том, каков общий объем и сколько свободного места осталось на каждом из смонтированных устройств.

Аргументы

Когда утилита df вызывается без аргументов, она сообщает о свободном пространстве на каждом из смонтированных на локальной системе устройств.

Список_файловых_систем — это необязательный список из одного или нескольких путевых имен, определяющих файловые системы, которые желательно включить в отчет. Эти аргументы работают на Mac OS X и на некоторых Linux-системах. Сослаться на смонтированную файловую систему можно по путевому имени ее устройства или по путевому имени каталога, на котором она смонтирована.

Ключи

Ключи, перед которыми ставится двойной дефис (`--`), работают только под Linux. За исключением особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--all -a` Вывод сведений о файловых системах с размером 0 блоков, например о /dev/proc. Обычно df о таких файловых системах не сообщает.

`--block-size=`***sz*** `-B` ***sz*** Аргумент ***sz*** определяет единицу измерения, используемую в отчете (по умолчанию это блоки размером 1 Кб), и является мультипликативным суффиксом из табл. V.1. См. также ключи `-h` (`--human-readable`) и `-H` (`--si`). (Linux)

`-g` (гигабайт). Вывод размеров в одногигабайтных блоках. (OS X)

`--si -H` Вывод размеров там, где это приемлемо, в K (килобайтных), M (мегабайтных) и G (гигабайтных) блоках. Использует степени числа 1000.

`--human-readable -h` Вывод размеров там, где это приемлемо, в K (килобайтных), M (мегабайтных) и G (гигабайтных) блоках. Использует степени числа 1024.

`--inodes -i` Вывод вместо отчета о блоках количества задействованных и свободных i-узлов.

`-k` (килобайт). Вывод размеров в однокилобайтных блоках.

`--local -l` Вывод информации только о локальных файловых системах.

`-m` (мегабайт). Вывод размеров в одномегабайтных блоках. (OS X)

`--type=`***fstype*** `-t` ***fstype*** Вывод информации только о файловых системах типа ***fstype***, например DOS или NFS. Для отчета о файловых системах нескольких типов ключ нужно повторить. (Linux)

`--exclude-type=`***fstype*** `-x` ***fstype*** Вывод информации только о файловых системах, не принадлежащих типу ***fstype***. (Linux)

Примечания

При работе под управлением Mac OS X утилита df поддерживает переменную среды окружения `BLOCKSIZE` (см. раздел «BLOCKSIZE» в начале главы) и игнорирует размер блоков меньше 512 байт или больше одного гигабайта. При работе под управлением Mac OS X подсчет задействованных и свободных i-узлов (ключ `-i`) на файловых системах HFS+ не имеет смысла. На этих файловых системах новые файлы могут создаваться при наличии для этих файловых систем свободного пространства.

Примеры

В следующих примерах утилита df выводит информацию о всех файловых системах, смонтированных на локальной системе:

```
$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda12             1517920     53264   1387548   4% /
/dev/hda1                15522      4846      9875  33% /boot
/dev/hda8              1011928    110268    850256  11% /free1
/dev/hda9              1011928     30624    929900   3% /free2
/dev/hda10             1130540     78992    994120   7% /free3
/dev/hda5              4032092   1988080   1839188  52% /home
/dev/hda7              1011928        60    960464   0% /tmp
/dev/hda6              2522048    824084   1569848  34% /usr
zach:/c                2096160   1811392    284768  86% /zach_c
zach:/d                2096450   1935097    161353  92% /zach_d
```

Затем утилита df вызывается с ключами -l и -h, генерируя удобный для чтения (human-readable) список локальных файловых систем. Размеры в этих листингах приводятся в мегабайтах и гигабайтах.

```
$ df -lh
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda12            1.4G   52M  1.3G   4% /
/dev/hda1              15M  4.7M  9.6M  33% /boot
/dev/hda8             988M  108M  830M  11% /free1
/dev/hda9             988M   30M  908M   3% /free2
/dev/hda10            1.1G   77M  971M   7% /free3
/dev/hda5             3.8G  1.9G  1.8G  52% /home
/dev/hda7             988M   60k  938M   0% /tmp
/dev/hda6             2.4G  805M  1.5G  34% /usr
```

В следующем примере, работающем только под Linux, выводится информация о разделе /free2 в мегабайтных блоках:

```
$ df -BM /free2
Filesystem           1M-blocks      Used Available Use% Mounted on
/dev/hda9                  988        30       908   3% /free2
```

В последнем примере, работающем только под Linux, выводится информация о файловой системе NFS в удобном для чтения виде:

```
$ df -ht nfs
Filesystem            Size  Used Avail Use% Mounted on
zach:/c               2.0G  1.7G  278M  86% /zach_c
zach:/d               2.0G  1.8G  157M  92% /zach_d
```

diff

Вывод различий между двумя текстовыми файлами.

diff [ключи] файл1 файл2
diff [ключи] файл1 каталог
diff [ключи] каталог файл2
diff [ключи] каталог1 каталог2

Утилита diff выводит построчные различия между двумя текстовыми файлами. По умолчанию diff выводит различия как инструкции, которые затем можно будет использовать при редактировании одного из файлов, чтобы сделать его таким же, как другой файл.

Аргументы

Аргументы ***файл1*** и ***файл2*** являются путевыми именами обычных текстовых файлов, с которыми работает утилита diff. Когда вместо ***файл2*** используется аргумент ***каталог***, diff ищет файл в каталоге с таким же именем, как ***файл1***. Точно так же утилита работает, когда ***каталог*** указывается вместо ***файл1***. Если указываются два аргумента ***каталог***, diff сравнивает файлы в ***каталоге1*** с файлами, имеющими такие же имена в ***каталоге2***.

Ключи

Утилита diff допускает применение общих ключей, описание которых дано в табл. V.2, с одним лишь исключением: когда один из аргументов является каталогом, а другой простым файлом, сравнение со стандартным вводом проводить нельзя.

ВЕРСИЯ DIFF ДЛЯ MAC OS X ДОПУСКАЕТ ИСПОЛЬЗОВАНИЕ ДЛИННЫХ КЛЮЧЕЙ

Ключи для diff, перед которыми ставится двойной дефис (– –), работают как под Mac OS X, так и под Linux.

`--ignore-blank-lines` `-B` — Игнорирование разницы, заключающейся только лишь в пустых строках.

`--ignore-space-change` `-b` — Игнорирование пробельных символов (пробелов и символов табуляции) в конце строк.

`--context` `[=`*строки*`]` `-C [`*строки*`]` — Вывод различающихся частей двух файлов, включая предыдущую и последующую ***строки*** (по умолчанию 3) для демонстрации контекста. Каждая строка в ***файле1***, отсутствующая в ***файле2***, предваряется дефисом (-), каждая лишняя строка в ***файле2*** предваряется знаком плюс (+), а строки, имеющие различные версии в обоих файлах, предваряются восклицательным знаком (!). Когда среди строк, составляющих окружающие ***строки***, попадаются различающиеся сами по себе строки, они в выводе группируются вместе.

`--ed` `-e` — Создание и отправка на стандартный вывод сценария для редактора **ed**, который будет редактировать ***файл1***, создавая из него подобие ***файла2***. В конце сценария, если планируется перенаправить ввод из сценария в **ed**, нужно добавить `w` (Write — записать) и `q` (Quit — завершить). При использовании ключа `--ed` утилита diff выводит изменения в обратном порядке: перед изменениями, внесенными в начало файла, выводятся изменения, внесенные в его конец, предохраняя более ранние изменения от влияния на более поздние, когда сценарий используется в качестве ввода в **ed**. Например, если строка ближе к началу была удалена, последующие номера строк в сценарии будут неправильными.

`--ignore-case` `-i` — Игнорирование разницы в регистрах букв при сравнении файлов.

`--new-file -N` Анализ файла, присутствующего при сравнении каталогов только в одном из них (в другом — в виде пустого файла).

`--show-c-function` `-p` Демонстрация, на какую функцию языка C, управляющую структуру bash, подпрограмму Perl и так далее, влияет каждое изменение.

`--brief` `-q` Отказ от вывода разницы между строками в файлах. Взамен diff выводит только отчет о том, что файлы отличаются друг от друга.

`--recursive` `-r` При использовании diff для сравнения файлов в двух каталогах принуждение к спуску сравнения по иерархиям каталогов.

`--unified [=`*строки*`]` `-U` *строки* Использование удобного для чтения унифицированного формата вывода. Более подробное описание и пример приведены в разборе утилиты diff в разделе «diff: Сравнение двух файлов» на с. 74. Аргумент ***строки*** задает количество строк окружающего контекста, которое по умолчанию равно трем.

`--ignore-all-space` `-w` (whitespace — пробельные символы). Игнорирование пробельных символов при сравнении строк.

`--width=`*n* `-W` *n* Установка ширины столбцов, которые используются утилитой diff для отображения вывода, в *n* символов. Этот ключ применяется с ключом `--side-by-side`. Утилита sdiff (см. раздел «Примечания») использует для выполнения такой же функции букву w в нижнем регистре: `-w` *n*.

`--side-by-side` `-y` Отображение вывода в формате параллельного расположения. Этот ключ приводит к генерации такого же вывода, как и у утилиты sdiff. С этим ключом следует использовать ключ `--width=`*n*.

Примечания

Утилита sdiff похожа на утилиту diff, но ее вывод легче читается. При использовании с ключом `--side-by-side` утилита diff отображает такой же вывод, что и утилита sdiff. Дополнительные сведения даны в разделе «Примеры» и в man- и info-страницах, посвященных утилитам diff и sdiff.

Для сравнения трех файлов следует использовать утилиту diff3.

Для сравнения нетекстовых (двоичных) файлов следует использовать утилиту cmp (см. соответствующий раздел).

Комментарии

Когда утилита diff используется без ключей, она выдает серию строк, содержащих инструкции Добавить — Add (`a`), Удалить — Delete (`d`) и Изменить — Change (`c`). За каждой из этих строк следуют строки файла, которые нужно соответственно добавить к нему, удалить из него или отредактировать, для того чтобы файлы стали одинаковыми. Символ «меньше чем» (<) ставится перед строками из ***файла1***. Символ «больше чем» (>) ставится перед строками из ***файла2***. Вывод утилиты diff появляется

в формате, показанном в табл. V.12. Пара номеров строк, разделенная запятой, представляет диапазон строк, а отдельный номер строки представляет собой одну строку.

Утилита diff предполагает, что вы собираетесь преобразовать ***файл1*** в ***файл2***. Номера строк слева от каждой инструкции a, c или d всегда относятся к ***файлу1***, а номера строк справа от инструкции относятся к ***файлу2***. Чтобы преобразовать файл2 в файл1, нужно еще раз запустить diff, поменяв местами аргументы.

Таблица V.12. Вывод утилиты diff

Инструкция	**Значение (для преобразования файла *файл1* в файл *файл2*)**
строка1 a строка2,строка3 > строки из файла ***файл2***	Добавление строк со строки строка2 и до строки строка3 из файла ***файл2*** после строки строка1 в файле ***файл1***
строка1,строка2 d строка3 < строки из файла ***файл1***	Удаление, начиная со строки строка1 и до строки строка2 из файла ***файл1***
строка1,строка2 c строка3,строка4 < строки из файла ***файл1*** --- > строки из файла ***файл2***	Замена строк со строки строка1 и до строки строка2 в файле ***файл1*** на строки со строки строка3 и до строки строка4 из ***файла2***

Примеры

В первом примере показано, как утилита diff выводит разницу между двумя короткими, похожими друг на друга файлами:

```
$ cat m
aaaaa
bbbbb
ccccc
$ cat n
aaaaa
ccccc
$ diff m n
2d1
< bbbbb
```

Разница между файлами m и n состоит в том, что вторая строка файла m (bbbbb) отсутствует в файле n. Первая строка, выводимая утилитой diff (2d1), показывает, что для того чтобы сделать файл по имени ***файл1*** (m) одинаковым с файлом по имени ***файл2*** (n), нужно удалить из него вторую строку. Следующая выведенная утилитой diff строка начинается с символа «меньше чем» (<), показывающего, что эта строка текста из файла по имени ***файл1***. В этом примере данная информация не нужна, все что нужно знать — номер строки, чтобы эту строку можно было удалить.

Если ширина вывода установлена в 30 символов, то как при использовании ключа --side-by-side, так и при использовании утилиты sdiff отображается один и тот же вывод. В этом выводе символ «меньше чем» указывает на дополнительную строку в файле m; утилита diff (sdiff) оставляет пустую строку в файле n, куда должна быть помещена дополнительная строка, чтобы файлы стали одинаковыми.

```
$ diff --side-by-side --width=30 m n
aaaaa                 aaaaa
bbbbb               <
```

```
ccccc           ccccc

$ sdiff -w 30 m n
aaaaa           aaaaa
bbbbb         <
ccccc           ccccc
```

В следующем примере используется такой же файл m и новый файл p, чтобы показать утилиту diff, выдающую инструкцию Добавить — a (Append):

```
$ cat p
aaaaa
bbbbb
rrrrr
ccccc
$ diff m p
2a3
> rrrrr
```

В предыдущем примере утилита diff выдает инструкцию 2a3, чтобы показать, что к файлу m следует добавить строку после строки 2, чтобы он стал таким же, как файл p. Вторую строку diff выводит, показывая, что она из файла p (строка начинается с символа >, свидетельствующего о ее принадлежности к файлу ***файл2***). В этом примере требуется информация о том, что эта строка должна содержать текст rrrrr.

В следующем примере, чтобы показать как утилита diff обозначает строку, требующую изменения, опять используется файл m, но на этот раз уже с файлом r:

```
$ cat r
aaaaa
-q
ccccc

$ diff m r
2c2
< bbbbb
---
> -q
```

Разница между двумя файлами проявляется в строке 2: файл m содержит bbbbb, а файл r содержит -q. Утилита diff выводит 2c2, чтобы показать, что нужно изменить строку 2. После обозначения требуемого изменения diff показывает, что нужно изменить строку 2 в файле m (bbbbb), превратив ее в строку 2 в файле r (-q), чтобы файлы стали одинаковыми. Три дефиса показывают конец того текста в файле m, который нуждается в изменении, и начало текста в файле r, которым его нужно заменить.

Сравнивание двух похожих файлов с использованием ключей для задания параллельного расположения и ширины (-y и -W) приводит к удобочитаемому результату. Символ канала (|) свидетельствует о том, что строка с одной стороны должна быть заменена строкой с другой стороны, чтобы файлы стали одинаковыми:

```
$ diff -y -W 30 m r
aaaaa           aaaaa
bbbbb         | -q
ccccc           ccccc
```

В следующих примерах сравниваются два файла q и v:

```
$ cat q          $ cat v
Monday           Monday
Tuesday          Wednesday
Wednesday        Thursday
Thursday         Thursday
Saturday         Friday
Sunday           Saturday
                 Sundae
```

При запуске в режиме параллельного отображения утилита diff показывает, что в файле v отсутствует строка Tuesday, а в файле q есть только одна строка Thursday (в файле v их две), а строка Friday отсутствует в файле q. В файле q последняя строка — Sunday, а в файле v последняя строка — Sundae: утилита diff показывает, что эти строки отличаются друг от друга. Можно изменить файл q, чтобы сделать его одинаковым с файлом v, удалив строку Tuesday, добавив по одной строке Thursday и Friday и подставив строку Sundae из файла v вместо строки Sunday в файле q. Можно также изменить файл v, чтобы сделать его одинаковым с файлом q, добавив строку Tuesday, удалив одну из строк Thursday и строку Friday и подставив строку Sunday из файла q вместо строки Sundae в файле v.

```
$ diff -y -W 30 q v
Monday                Monday
Tuesday             <
Wednesday             Wednesday
Thursday              Thursday
                    > Thursday
                    > Friday
Saturday              Saturday
Sunday              | Sundae
```

Context diff

С ключом --context (и названием *context diff* — контекстная разница) утилита diff отображает вывод, сообщающий о том, как превратить первый файл во второй. В двух верхних строках идентифицируются файлы и показывается, что q представляется звездочками, а v — дефисами. Сразу за строкой звездочек, являющихся индикатором начала фрагмента текста, следует строка звездочек с числами 1,6 посередине. Эта строка показывает, что инструкции в первом разделе предписывают, что нужно удалить из файла q или изменить в его содержимом, а именно строки с 1 по 6 (то есть все строки файла q; в более длинных файлах эта строка будет обозначать первый фрагмент). Дефис во второй из следующих строк показывает, что нужно удалить строку со словом Tuesday. Строка с восклицательным знаком показывает, что нужно заменить строку со словом Sunday соответствующей строкой из файла v. Строка, состоящая из дефисов с числами 1,7 посередине, показывает, что в следующем разделе будет дано предписание, какие строки из файла v, строки с 1 по 7, нужно добавить или изменить в файле q. Нужно добавить вторую строку со словом Thursday и строку со словом Friday и нужно изменить Sunday в файле q на Sundae (из файла v).

```
$ diff --context q v
*** q   Mon Aug 24 18:26:45 2009
--- v   Mon Aug 24 18:27:55 2009
```

```
***************
*** 1,6 ****
  Monday
- Tuesday
  Wednesday
  Thursday
  Saturday
! Sunday
--- 1,7 ----
  Monday
  Wednesday
  Thursday
+ Thursday
+ Friday
  Saturday
! Sundae
```

diskutil (OS X)

Проверка, изменение и восстановление локальных томов.

diskutil ***действие [аргументы]***

Утилита diskutil осуществляет монтирование, размонтирование и вывод информации о дисках и разделах (томах). Она может также форматировать и восстанавливать файловые системы и разбивать диск на разделы.

Аргументы

Действие определяет, что нужно делать утилите diskutil. В табл. V.13 перечислены наиболее распространенные ***действия***, а также используемые ими аргументы.

Таблица V.13. Действия утилиты diskutil

Действие	Аргумент	Описание
eraseVolume	*тип имя устройство*	Переформатирование устройства с использованием *типа* формата и метки *имени*. *Имя* указывает на имя тома; проще всего работать с именами, состоящими из букв и цифр. Обычно в качестве *типа* файловой системы используется HFS+, но это также может быть UFS или MS-DOS. В качестве части *типа* можно указать дополнительные ключи. Например, файловая система FAT32 (которая используется в Windows 98 и более поздних версиях) обозначается *типом* MS-DOS FAT32. Чувствительная к регистру символов файловая система HFS+ будет иметь *тип* Case-sensitive Journaled HFS+
info	*Устройство*	Вывод информации об *устройстве*. Не требует быть владельцем *устройства*
list	[*Устройство*]	Вывод списка разделов *устройства*. Без аргумента *устройство* вывод разделов, имеющихся на всех устройствах. Не требует быть владельцем *устройства*
mount	*Устройство*	Монтирование *устройства*

Действие	Аргумент	Описание
mountDisk	*Устройство*	Монтирование всех устройств на диске, содержащем *устройство*
reformat	*Устройство*	Переформатирование *устройства* с использованием его текущего имени и формата
repairVolume	*Устройство*	Восстановление файловой системы на *устройстве*
unmount	*Устройство*	Размонтирование *устройства*
unmountDisk	*Устройство*	Размонтирование всех устройств на диске, содержащем *устройство*
verifyVolume	*Устройство*	Проверка файловой системы на *устройстве*. Не требует быть владельцем *устройства*

Примечания

Утилита diskutil предоставляет доступ к инфраструктуре управления диском — Disk Management framework, программе поддержки, используемой приложением Disk Utility. Она позволяет выбирать ряд действий, не поддерживаемых при работе в графическом интерфейсе.

При указании ***действия***, изменяющего состояние тома, вы должны быть владельцем ***устройства*** или работать с root-правами.

fsck На Linux-системах ***действия*** verifyVolume и repairVolume утилиты diskutil являются функциональными аналогами утилиты fsck. При работе под управлением OS X применение утилиты fsck не рекомендуется, за исключением тех случаев, когда система работает в однопользовательском режиме.

disktool Некоторые из функций, выполняемых утилитой diskutil, ранее выполнялись утилитой disktool.

Примеры

В первом примере выводится список дисковых устройств и томов, доступных на локальной системе:

```
$ diskutil list
/dev/disk0
   #:                       type name                 size      identifier
   0: Apple_partition_scheme                          *152.7 GB disk0
   1:    Apple_partition_map                          31.5 KB   disk0s1
   2:              Apple_HFS Eva01                    30.7 GB   disk0s3
   3:              Apple_HFS Users                    121.7 GB  disk0s5
/dev/disk1
   #:                       type name                 size      identifier
   0: Apple_partition_scheme                          *232.9 GB disk1
   1:    Apple_partition_map                          31.5 KB   disk1s1
   2:         Apple_Driver43                          28.0 KB   disk1s2
   3:         Apple_Driver43                          28.0 KB   disk1s3
   4:       Apple_Driver_ATA                          28.0 KB   disk1s4
```

```
   5:      Apple_Driver_ATA                    28.0 KB    disk1s5
   6:        Apple_FWDriver                    256.0 KB   disk1s6
   7:    Apple_Driver_IOKit                    256.0 KB   disk1s7
   8:         Apple_Patches                    256.0 KB   disk1s8
   9:             Apple_HFS Spare              48.8 GB    disk1s9
  10:             Apple_HFS House              184.1 GB   disk1s10
```

В следующем примере выводится информация об одной из смонтированных томов:

```
$ diskutil info disk1s9
   Device Node:        /dev/disk1s9
   Device Identifier:  disk1s9
   Mount Point:        /Volumes/Spare
   Volume Name:        Spare
   File System:        HFS+
   Owners:             Enabled
   Partition Type:     Apple_HFS
   Bootable:           Is bootable
   Media Type:         Generic
   Protocol:           FireWire
   SMART Status:       Not Supported
   UUID:               C77BB3DC-EFBB-30B0-B191-DE7E01D8A563

   Total Size:         48.8 GB
   Free Space:         48.8 GB

   Read Only:          No
   Ejectable:          Yes
```

В следующем примере форматируется раздел **/dev/disk1s8** под файловую систему HFS+ Extended (HFSX), и ему присваивается метка Spare2. Эта команда удаляет все данные раздела:

```
# diskutil eraseVolume 'Case-sensitive HFS+' Spare2 disk1s8
Started erase on disk disk1s10
Erasing
Mounting Disk
Finished erase on disk disk1s10
```

В последнем примере показан вывод успешной операции по проверке тома — verifyVolume:

```
$ diskutil verifyVolume disk1s9
Started verify/repair on volume disk1s9 Spare
Checking HFS Plus volume.
Checking Extents Overflow file.
Checking Catalog file.
Checking Catalog hierarchy.
Checking volume bitmap.
Checking volume information.
The volume Spare appears to be OK.
Mounting Disk
Verify/repair finished on volume disk1s9 Spare
```

ditto (OS X)

Копирование файлов и создание незапакованных архивов.

*ditto [ключи] **файл-источник файл-приемник***
*ditto [ключи] **список_файлов-источников каталог-приемник***
*ditto –c [ключи] **каталог-источник архив-приемник***
*ditto –x [ключи] **список_архивов-источников каталог-приемник***

Утилита ditto копирует файлы и их атрибуты принадлежности, времени и так далее, включая расширенные атрибуты. Копирование может осуществляться в архивные файлы, созданные утилитами cpio и zip, и из этих файлов. Утилита также может копировать обычные файлы и каталоги.

Аргументы

Файл-источник — это путевое имя файла, копируемого утилитой ditto. ***Файл-приемник*** — это путевое имя, присваиваемое утилитой ditto создаваемой копии файла.

В ***списке_файлов-источников*** указывается одно или несколько путевых имен файлов и каталогов, с которых ditto делает копии. ***Каталог-приемник*** — это путевое имя каталога, в который ditto копирует файлы и каталоги. Когда указывается ***каталог-приемник***, ditto дает каждому скопированному файлу такое же обычное имя файла, которое было у ***файла-источника***.

Каталог-источник — это отдельный каталог, который ditto копирует в ***архив-приемник***. Создаваемый архив хранит копии содержимого ***каталога-источника***, но не самого каталога.

В ***списке_архива-источника*** указывается одно или несколько путевых имен архивов, из которых ditto извлекает содержимое в ***каталог-приемник***.

Использование дефиса (-) вместо имени файла или каталога заставляет ditto считывать данные со стандартного ввода или записывать их на стандартный вывод, вместо чтения их из файла или каталога и записи в файл или каталог.

Ключи

Ключи -c и -x вместе не используются.

`-c` (create archive — создать архив). Создание архива файлов.

`--help` Вывод справочного сообщения.

`-k` (pkzip). Использование zip-формата вместо используемого по умолчанию формата cpio (см. соответствующий раздел) для создания или извлечения архивов. Дополнительные сведения о zip даны в главе 3, в совете «Сравнение gzip и zip» раздела «gzip: Сжатие файла» на с. 82.

`--norsrc` (no resource — без ресурсов). Игнорирование расширенных атрибутов. Этот ключ заставляет ditto копировать только ветви данных файла — data forks (при работе под управлением Mac OS X 10.3 и более ранних версий это делается по умолчанию).

`--rsrc` (resource — с ресурсами). Копирование расширенных атрибутов, включая ветви ресурсов — resource forks (при работе под управлением Mac OS X 10.3 и более поздних версий это делается по умолчанию). Используются также ключи `-rsrc` и `-rsrcFork`.

`-V` (very verbose — очень подробно). Отправка строки на стандартную ошибку для каждого файла, символьной ссылки и узла устройства, копируемого ditto.

`-v` (verbose — подробно). Отправка строки на стандартную ошибку для каждого копируемого ditto каталога.

`-X` (exclude — исключение). Удерживание ditto от поиска в каталогах, принадлежащих файловым системам, отличающимся от той, которая содержит файлы, конкретно указанные для копирования.

`-x` (extract archive — извлечение из архива). Извлечение файлов из архивного файла.

`-z` (compress — сжатие). Использование gzip (см. соответствующий раздел) или gunzip для сжатия и распаковки архивов cpio.

Примечания

Утилита ditto не копирует флаг атрибута блокировки. Утилита также не копирует ACL-списки.

По умолчанию ditto создает и читает архивы в формате cpio (см. соответствующий раздел).

Утилита ditto не может выводить список содержимого архивных файлов, она может только создавать архивы и извлекать из них файлы. Для вывода списка содержимого архивов в формате cpio следует использовать утилиту pax или утилиту cpio, а для вывода содержимого zip-файлов следует использовать утилиту unzip с ключом `-l`.

Примеры

В следующих примерах показаны три способа резервного копирования личного каталога пользователя, включая расширенные атрибуты (с извлечениями, упомянутыми в разделе «Примечания») и сохранение показателей времени и прав доступа. В первом примере личный каталог пользователя Zach копируется на том (файловую систему), названный Backups, копия является новым каталогом по имени zach.0228:

```
$ ditto /Users/zach /Volumes/Backups/zach.0228
```

В следующем примере личный каталог пользователя Zach копируется в отдельный архивный файл в формате cpio на томе по имени Backups:

```
$ ditto -c /Users/zach /Volumes/Backups/zach.0228.cpio
```

В следующем примере личный каталог пользователя Zach копируется в zip-архив:

```
$ ditto -c -k /Users/zach /Volumes/Backups/zach.0228.zip
```

В каждом из трех следующих примеров в личный каталог пользователя Zach восстанавливается соответствующий архив резервных копий файлов, при этом перезаписываются все уже имеющиеся там файлы:

```
$ ditto /Volumes/Backups/zach.0228 /Users/zach
$ ditto -x /Volumes/Backups/zach.0228.cpio /Users/zach
$ ditto -x -k /Volumes/Backups/zach.0228.zip /Users/zach
```

В следующем примере каталог Scripts копируется в каталог по имени Scripts-Backups на удаленном хосте bravo. В нем используется аргумент-дефис вместо локального ***архива-приемника*** для записи на стандартный вывод и вместо ***списка_архивов-источников*** на удаленной системе для чтения со стандартного ввода:

```
$ ditto -c Scripts - | ssh bravo ditto -x - ScriptsBackups
```

В последнем примере локальный загрузочный диск (корневая файловая система) копируется на том по имени Backups.root. Поскольку некоторые из файлов могут быть прочитаны только привилегированным пользователем, сценарий должен быть запущен пользователем с root-правами. Ключ `-X` заставляет ditto обойтись без попыток копирования других томов (файловых систем) смонтированных ниже корневого каталога /.

```
# ditto -X / /Volumes/Backups.root
```

dmesg

Вывод сообщений ядра.

*dmesg [**ключи**]*

Утилита dmesg выводит сообщения, хранящиеся в кольцевом буфере ядра.

Ключи

`-c` Очистка кольцевого буфера ядра после запуска dmesg. (Linux)

`-M` *core* Аргумент ***core*** является именем обрабатываемого файла (дампа памяти), по умолчанию используется /dev/kmem. (OS X)

`-N` *kernel* Аргумент ***kernel*** является путевым именем файл ядра (по умолчанию — /mach). Если выводится информация о дампе памяти, ***kernel*** должен быть ядром, которое работало во время создания файла ***core***. (OS X)

Комментарии

При запуске системы ядро заполняет свой кольцевой буфер сообщениями, относящимися к инициализации оборудования и модулей. Сообщения в кольцевом буфере ядра зачастую используются для диагностики системных проблем.

Примечания

При работе под управлением Mac OS X нужно запускать эту утилиту, обладая root-правами.

Имея кольцевое устройство, буфер сообщений ядра хранит самые последние полученные им сообщения, затирая по мере заполнения самые старые сообщения. Чтобы

сохранить список сообщений ядра, выданных при загрузке, нужно сразу же после загрузки системы и входа в нее ввести следующую команду:

```
$ dmesg > dmesg.boot
```

Эта команда сохранит сообщения ядра в файле dmesg.boot. Список этих сообщений может быть весьма познавательным и очень полезным при возникновении проблем в процессе загрузки системы.

При работе с большинством Linux-систем после загрузки системы системные записи в файле /var/log/messages или в ему подобном файле содержат во многом такую же информацию, которую выводит утилита dmesg.

Примеры

Следующая команда выводит сообщения ядра, находящиеся в кольцевом буфере и содержащие строку serial, независимо от регистра символов:

```
$ dmesg | grep -i serial
Apple16X50PCI2: Identified 4 Serial channels at PCI SLOT-2 Bus=5 Dev=2 Func=0
```

dscl (OS X)

Вывод информации и управление Службой каталогов (Directory Service).

dscl [ключи] [источник_данных [команда]]

Утилита dscl (Directory Service command line — командная строка Службы каталогов) позволяет работать с узлами каталогов Службы каталогов. При вызове без аргументов утилита dscl работает в интерактивном режиме.

Аргументы

Источник_данных — это имя узла или хоста Mac OS X Server, указанное в виде имени хоста или IP-адреса. Точка (.) указывает на локальный домен.

Ключи

-p (prompt — приглашение). Вывод приглашения на ввод пароля, если это необходимо.

-q (quiet — молчание) Отказ от вывода приглашения.

-u *пользователь* Аутентификация в качестве пользователя.

Команды

Определение некоторых, используемых здесь терминов дается в разделе «Примечания».

Дефис (-) перед командой ставить необязательно.

-list *путь* [*ключ*] (также -ls). Построчный вывод списка подкаталогов в ***пути***. При указании ***ключа*** эта команда выводит список каталогов, соответствующих ***ключу***.

`-read [`*`путь`* `[`*`ключ`*`]]`	(также `-cat` и `.`). Построчный вывод свойств каталога.
`-readall [`*`путь`* `[`*`ключ`*`]]`	Вывод свойств с заданным ***ключом***.
`-search` *`путь ключ значение`*	Вывод свойств, в которых ***ключ*** соответствует ***значению***.

Примечания

При рассмотрении Службы каталогов термин ***каталог*** относится к коллекции данных (базе данных), а не к каталогу файловой системы. В каждом каталоге хранится одно или несколько свойств. Каждое свойство содержит пару ключ-значение, где для каждого заданного ключа может быть более одного значения. Чаще всего утилита dscl выводит свойство, сначала показывая ключ, за которым следует двоеточие, а затем значение. Если имеется более одного значения, они разделяются пробелами. Если в значении имеются пробелы, утилита dscl выводит значение в строке, следующей за ключом.

При работе под управлением Mac OS X и Mac OS X Server в Open Directory хранится информация для локальной системы в XML-файлах, имеющих формат ключ-значение и находящихся в иерархии каталога /var/db/dslocal.

Утилита dscl является работающем в режиме командной строки эквивалентом средства NetInfo Manager (которое было доступно в версиях Mac OS X, вплоть до 10.5) или средства Workgroup Manager на Mac OS X Server.

Примеры

Команда `dscl -list`, если указать путь /, выводит список каталогов верхнего уровня:

```
$ dscl . -list /
AFPServer
AFPUserAliases
Aliases
AppleMetaRecord
Augments
Automount
…
SharePoints
SMBServer
Users
WebServer
```

Точка, использованная в качестве первого аргумента для dscl, определяет источником данных локальный домен. Следующая команда выводит список каталогов Users:

```
$ dscl . -list /Users
_amavisd
_appowner
_appserver
_ard
…
```

```
_www
_xgridagent
_xgridcontroller
daemon
max
nobody
root
```

Команду `dscl -read` можно использовать для вывода информации об определенном пользователе:

```
$ dscl . -read /Users/root
AppleMetaNodeLocation: /Local/Default
GeneratedUID: FFFFEEEE-DDDD-CCCC-BBBB-AAAA00000000
NFSHomeDirectory: /var/root
Password: *
PrimaryGroupID: 0
RealName:
 System Administrator
RecordName: root
RecordType: dsRecTypeStandard:Users
UniqueID: 0
UserShell: /bin/sh
```

Следующая команда `dscl -readall` выводит список всех имен пользователей и пользовательские ID на локальной системе. Команда ищет ключи `RecordName` и `UniqueID` в каталоге **/Users** и выводит связанные с ними значения. Утилита **dscl** отделяет несколько значений друг от друга пробелами.

```
$ dscl . -readall /Users RecordName UniqueID
RecordName: _amavisd amavisd
UniqueID: 83
-
RecordName: _appowner appowner
UniqueID: 87
-
…
RecordName: daemon
UniqueID: 1
-
RecordName: sam
UniqueID: 501
-
RecordName: nobody
UniqueID: -2
-
RecordName: root
UniqueID: 0
```

В следующем примере для вывода всех свойств, у которых ключ `RecordName` имеет значение `sam`, используется команда `dscl -search`:

```
$ dscl . -search / RecordName sam
Users/sam               RecordName = (
    sam
)
```

du

Вывод информации об использовании диска иерархией каталога и (или) файлом.

du [ключи] [список_путей]

Утилита du (disk usage — использование диска) сообщает о количестве дискового пространства, занятого иерархией каталога или файлом. По умолчанию du выводит количество 1024-байтовых блоков, занятых иерархией каталога или файлом.

Аргументы

Без аргументов утилита du выводит информацию о рабочем каталоге и его подкаталогах. Аргумент ***список_путей*** определяет те каталоги и файлы, о которых нужно получить информацию.

Ключи

Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

Без ключей утилита du выводит общее количество пространства накопителя, занятого каждым объектом из ***списка_путей***. Для каталогов утилита du выводит это общее количество занятого пространства после рекурсивного вывода количества общего пространства, занятого каждым подкаталогом.

`--all` `-a` Вывод количества пространства, занятого всеми обычными файлами наряду о количеством общего пространства, занятого каждым каталогом.

`--block-size=`*sz* `-B` *sz* Аргумент ***sz*** определяет единицу измерения, используемую в отчете. Он является мультипликативным суффиксом из табл. V.1. См. также ключи `-H` (`--si`) и `-h` (`--human-readable`). (Linux)

`--total -c` Отображение в конце вывода итогового количества занятого пространства.

`--dereference-args` `-D` (с частичным разыменованием) Для каждого файла, являющегося символьной ссылкой, вывод отчета о файле, на который указывает ссылка, а не о самой символьной ссылке. Действие этого ключа распространяется на файлы, указанные в командной строке, оно не касается файлов, найденных при спуске по иерархии каталога. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. (Linux).

`-d` *глубина* Вывод информации о подкаталогах для уровня каталогов, указанного аргументом ***глубина***. (OS X)

`--si -H` (human readable — удобный для чтения). Вывод размеров там, где это приемлемо, в K (килобайтных), M (мегабайтных) и G (гигабайтных) блоках. Использует степени числа 1000. (Linux)

`-H` (с частичным разыменованием). Для каждого файла, являющегося символьной ссылкой, вывод отчета о файле, на который указывает ссылка, а не о самой

символьной ссылке. Действие этого ключа распространяется на файлы, указанные в командной строке, оно не касается файлов, найденных при спуске по иерархии каталога. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L». В будущем ключ `-H` будет изменен и станет эквивалентом ключа `-D`. (OS X)

`--human-readable` `-h` Вывод размеров там, где это приемлемо, в K (килобайтных), M (мегабайтных) и G (гигабайтных) блоках. Использует степени числа 1024.

`-k` Вывод размеров в 1-килобайтных блоках.

`--dereference` `-L` Для каждого файла, являющегося символьной ссылкой, вывод отчета о файле, на который указывает ссылка, а не о самой символьной ссылке. Действие этого ключа распространяется на все файлы. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. По умолчанию используется ключ `-P` (`--no-dereference`). Пример его использования приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L».

`-m` Вывод размеров в 1-мегабайтных блоках.

`--no-dereference` `-P` Для каждого файла, являющегося символьной ссылкой, вывод отчета о символьной ссылке, а не о файле, на который она указывает. Действие этого ключа распространяется на все файлы. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Эти действия осуществляются по умолчанию. Пример использования этого ключа приведен в разделе «chgrp», в подразделе «Примеры».

`--summarize` `-s` Вывод только общего количества для каждого каталога или файла, указанного в командной строке, общее количество для подкаталогов не выводится.

`--one-file-system -x` Вывод отчета только о тех файлах и каталогах, которые относятся к той же файловой системе, что и обрабатываемый аргумент.

Примеры

В первом примере утилита du выводит информацию о размере подкаталогов в рабочем каталоге. В последней строке содержатся итоговые данные о рабочем каталоге и его подкаталогах.

```
$ du
26      ./Postscript
4       ./RCS
47      ./XIcon
4       ./Printer/RCS
12      ./Printer
105     .
```

Итоговые данные (105) — это количество блоков, занятых всеми обычными файлами и каталогами, находящимися ниже рабочего каталога. Ведется подсчет пространства, занятого всеми файлами, даже если du выводит только размеры каталогов.

Если отсутствует право на чтение файла или каталога, попадающего в подсчет утилиты du, она отправляет на стандартную ошибку предупреждение и пропускает этот файл или каталог. Затем, используя ключ -s (summarize — подведение итога), du выводит итоговое значение для каждого каталога в /usr, но не выводит информацию для их подкаталогов:

```
$ du -s /usr/*
4       /usr/X11R6
260292  /usr/bin
10052   /usr/games
7772    /usr/include
1720468 /usr/lib
105240  /usr/lib32
0       /usr/lib64
du: cannot read directory '/usr/local/lost+found': Permission denied
  du: невозможно прочесть каталог '/usr/local/lost+found': Отсутствие прав доступа
…
130696  /usr/src
```

При добавлении к предыдущему примеру ключа -c (total) du выводит такой же список, но с итоговыми сведениями в конце:

```
$ du -sc /usr/*
4       /usr/X11R6
260292  /usr/bin
…
130696  /usr/src
3931436 total
```

В следующем примере используются ключи -s (summarize), -h (human-readable) и -c (total):

```
$ du -shc /usr/*
4.0K    /usr/X11R6
255M    /usr/bin
9.9M    /usr/games
7.6M    /usr/include
1.7G    /usr/lib
103M    /usr/lib32
…
128M    /usr/src
3.8G    total
```

В последнем примере в удобном для чтения формате выводится общий размер всех файлов, которые пользователь может прочитать в файловой системе /usr. Перенаправление стандартной ошибки в /dev/null отбрасывает все предупреждения о нечитаемых файлах и каталогах.

```
$ du -hs /usr 2>/dev/null
3.8G    /usr
```

echo

Вывод сообщения.

echo [ключи] сообщение

Утилита echo копирует свои аргументы, завершая их разделителем строк, в стандартный вывод. У каждой из оболочек Bourne Again и TC Shells имеется своя собственная версия встроенной команды, работающей аналогично утилите echo.

Аргументы

Сообщение состоит из одного или нескольких аргументов, которые могут включать в себя строки, взятые в кавычки, нестрогие ссылки на файлы и переменные оболочки. Аргументы отделяются друг от друга пробелами. Оболочка различает в аргументах непроцитированные специальные символы. Например, оболочка расширяет звездочку в список имен файлов, имеющихся в рабочем каталоге.

Ключи

Встроенную в оболочку tcsh команду echo можно настроить на разное рассмотрение эскейп-последовательностей, использующих обратный слэш, и ключа -n. Подробности даны по ссылке echo_style в man-странице, посвященной tcsh. При обычной настройке tcsh распознает ключ -n, разрешает использование эскейп-последовательностей, использующих обратный слэш, и игнорирует ключи -E и -e.

-E Подавление интерпретации эскейп-последовательностей, использующих обратный слэш, например \n. Этот ключ доступен только в версии встроенной команды оболочки bash.

-e Разрешение интерпретации эскейп-последовательностей, использующих обратный слэш, например \n. Этот ключ доступен только в версии встроенной команды оболочки bash.

--help Предоставление краткой справки по использованию утилиты echo. В нее включается список эскейп-последовательностей, использующих обратный слэш, интерпретируемых утилитой echo. Этот ключ работает только с утилитой echo и не работает с одноименными встроенными командами. (Linux)

-n Подавление разделителя строк, завершающего ***сообщение***.

Примечания

Подавление интерпретации эскейп-последовательностей, использующих обратный слэш, используется по умолчанию в bash-версии встроенной команды echo и в утилите echo.

Утилиту echo можно использовать для отправки сообщений на экран из сценария оболочки. Способы использования echo для вывода имен файлов с применением групповых символов рассмотрены в главе 5, в разделе «Специальный символ „*“» на с. 158.

Утилита echo и одноименные встроенные команды предоставляют в сообщениях эскейп-нотацию для представления некоторых, не выводимых на печать символов (табл. V.14). Чтобы эти эскейп-последовательности, использующие обратный слэш, работали с утилитой echo и встроенной в bash командой echo, следует воспользоваться ключом -e. А при использовании встроенной в tcsh команды echo ключ -e обычно не требуется.

Таблица V.14. Эскейп-последовательности, использующие обратный слэш

Последовательность	Значение
\a	Звонок
\c	Подавление замыкающего разделителя строк
\n	Разделитель строк
\t	Символ горизонтальной табуляции
\v	Символ вертикальной табуляции
\\	Обратный слэш

Примеры

В этом разделе приводятся лишь некоторые команды, использующие echo. Эти команды будут работать с утилитой echo (/bin/echo), а также с командами echo, встроенными в оболочки bash и tcsh, за исключением последней, для которой под управлением оболочки tcsh ключ -e может не пригодиться.

```
$ echo "Эта команда выводит строку."
Эта команда выводит строку.
$ echo -n "Эта отображаемая строка не сопровождается разделителем строк."
Эта отображаемая строка не сопровождается разделителем строк.$ echo hi
hi
$ echo -e "Это сообщение содержит\v вертикальную табуляцию."
Это сообщение содержит
                    вертикальную табуляцию.
$
```

В следующих примерах содержатся сообщения с эскейп-последовательностью, использующей обратный слэш \c. В первом примере оболочка обрабатывает аргументы для вызова echo. Когда оболочка встречает \c, она заменяет \c символом c. В следующих трех примерах показано, как процитировать \c, чтобы оболочка передавала эту последовательность утилите echo, которая затем не добавляла разделитель строк к концу сообщения. Первые четыре примера запускаются под управлением bash и требуют использования ключа -e. Последний пример запускается под управлением оболочки tcsh, для которой этот ключ может не понадобиться.

```
$ echo -e После этой строки следует разделитель строк.\c
После этой строки следует разделитель строк.c
$ echo -e 'После этой строки нет разделителя строк.\c'
После этой строки нет разделителя строк.$

$ echo -e "После этой строки нет разделителя строк.\c"
There is no newline after this line.$

$ echo -e После этой строки нет разделителя строк.\\c
После этой строки нет разделителя строк.$

$ tcsh
tcsh $ echo -e 'После этой строки нет разделителя строк.\c'
После этой строки нет разделителя строк.$
```

Вместо -e и \c можно использовать ключ -n.

expr

Вычисление выражения.

expr ***выражение***

Утилита expr вычисляет выражение и отправляет результат на стандартный вывод. Она вычисляет строки символов, представляющих либо числовые, либо нечисловые значения. Чтобы построить выражения, наряду со строками используются операторы.

Аргументы

Выражение состоит из строк, между которыми вставляются операторы. Каждая строка и оператор составляет отдельный аргумент, который нужно отделить от других аргументов пробелом. Операторы, имеющие для оболочки специальное значение (например, оператор умножения *), должны быть процитированы.

Следующий перечень операторов утилиты expr представлен в порядке убывания приоритетности. Каждый оператор в составе группы операторов имеет одинаковый уровень приоритета. Порядок вычислений можно изменить путем использования круглых скобок.

: (сравнение). Сравнение двух строк, начиная с первого символа каждой строки и заканчивая последним символом второй строки. Вторая строка является регулярным выражением с предполагаемым знаком вставки (^) в качестве ее первого символа. Если утилита expr находит соответствие, она выводит количество символов во второй строке. Если expr не находит соответствие, она выводит нуль.

* (умножение)

/ (деление)

% (остаток от деления)

Работают только со строками, содержащими цифры от 0 до 9 и дополнительно лидирующий знак минуса. Преобразуют строки в целые числа, выполняют над числами указанные действия и осуществляют обратное преобразование результата в строку перед его отправкой на стандартный вывод.

\+ (сложение)

\- (вычитание)

Работают так же, как и предыдущая группа операторов.

< (меньше чем)

<= (меньше чем или равно)

= *или* == (равно)

!= (не равно)

>= (больше чем или равно)

> (больше чем)

Операторы отношений работают как с числовыми, так и нечисловыми аргументами. Если один или оба аргумента являются нечисловыми, проводится нечисловое сравнение с использованием машинной схемы упорядочения (обычно это ASCII-таблица).

Если оба аргумента являются числовыми, проводится сравнение чисел. Утилита expr выводит 1 (единицу), если сравнение дало истинный результат, и 0 (нуль) при ложном результате.

`&` (И)

Вычисление обоих аргументов. Если ни один из них не вычисляется в 0 или не является пустой строкой, expr выводит значение первого элемента. В противном случае она выводит 0 (нуль). Этот оператор требует цитирования.

`|` (ИЛИ)

Вычисление первого аргумента. Если он не вычисляется в 0 или в пустую строку, expr выводит значение первого аргумента. В противном случае она выводит значение второго аргумента. Этот оператор требует цитирования.

Примечания

Утилита expr возвращает статус выхода 0 (нуль), если выражение вычисляется во что-нибудь, отличающееся от пустой строки или числа 0, статус выхода 1, если выражение вычисляется в пустую строку или 0, и статус выхода 2, если выражение имеет неверный формат.

Хотя в описании утилиты и при ее работе expr делается различие между числовыми и нечисловыми аргументами, все аргументы этой утилиты являются нечисловыми (строками символов). Там где это имеет смысл, expr пытается преобразовать аргумент в число (например, при использовании оператора +). Если строка содержит символы, отличные от цифр от 0 до 9 и дополнительного лидирующего знака минуса, то утилита expr ее преобразовать не может. В частности, если строка содержит знак плюса или десятичную точку, expr считает ее нечисловой. Если оба аргумента считаются числовыми, делается сравнение чисел. Если один из них считается нечисловым, производится лексикографическое сравнение.

Примеры

В следующих примерах утилита expr производит вычисления, используя константы. Эту утилиту можно также использовать для вычислений в сценариях оболочки с использованием переменных. Четвертая команда выводит сообщение об ошибке из-за наличия недопустимой десятичной точки:

```
$ expr 17 + 40
57
$ expr 10 - 24
-14
$ expr -17 + 20
3
$ expr 5.3 + 4
expr: non-numeric argument  expr: нечисловой аргумент
```

Дополнительные арифметические возможности предоставляются операторами умножения (*), деления (/) и получения остатка от деления (%). Оператор умножения нуждается в цитировании (установке перед ним символа обратного слэша), чтобы оболочка не рассматривала его в качестве специального символа (нестрогой ссылки на

файлы). Ставить кавычки вокруг всего выражения нельзя, поскольку каждая строка *и оператор должны фигурировать в качестве отдельных аргументов.*

```
$ expr 5 \* 4
20
$ expr 21 / 7
3
$ expr 23 % 7
2
```

В следующих двух примерах показано, как круглые скобки изменяют порядок вычислений. Каждую круглую скобку нужно цитировать, а вокруг комбинации из скобки и обратного слэша ставить пробелы:

```
$ expr 2 \* 3 + 4
10
$ expr 2 \* \( 3 + 4 \)
14
```

Для определения отношений между числовыми и нечисловыми аргументами можно использовать операторы отношений. В следующих командах сравниваются две строки с целью определить их равенство друг другу, при этом expr выводит 0, когда это отношение является ложным, и 1, если оно является истинным.

```
$ expr fred = sam
0
$ expr sam = sam
1
```

В следующих примерах операторы отношений, которые должны быть процитированы, устанавливают порядок между числовыми и нечисловыми аргументами. В данном случае опять, если отношение является истиной, expr выводит 1.

```
$ expr fred \> sam
0
$ expr fred \< sam
1
$ expr 5 \< 7
1
```

В следующей команде 5 сравнивается с m. Если один из аргументов, сравниваемых expr с использованием оператора отношения является нечисловым, expr предполагает, что и второй оператор является нечисловым. В данном случае, поскольку m является нечисловым аргументом, expr рассматривает 5 как нечисловой аргумент. Операция сравнения проводится между ASCII (на многих системах) значениями m и 5. ASCII-значение m равно 109, а значение 5 равно 53, поэтому expr вычисляет отношение как истинное.

```
$ expr 5 \< m
1
```

В следующем примере оператор соответствия определяет, что четыре символа во второй строке соответствуют первым четырем символам в первой строке. Утилита expr выводит количество соответствующих символов (4).

```
$ expr abcdefghijkl : abcd
4
```

Оператор & выводит 0, если один или оба его аргумента являются нулями или пустыми строками; в противном случае он выводит первый аргумент:

```
$ expr '' \& book
0

$ expr magazine \& book
magazine

$ expr 5 \& 0
0

$ expr 5 \& 6
5
```

Оператор | выводит первый аргумент, если он не равен нулю или пустой строке; в противном случае он выводит второй аргумент:

```
$ expr '' \| book
book

$ expr magazine \| book
magazine

$ expr 5 \| 0
5

$ expr 0 \| 5
5

$ expr 5 \| 6
5
```

file

Вывод классификации файла.

file [ключ] список_файлов

Утилита file классифицирует файлы в соответствии с их содержимым.

Аргументы

Список_файлов — это список путевых имен одного или нескольких файлов, классифицируемых утилитой file. В ***списке_файлов*** можно указать файл любого типа, включая обычные файлы, каталоги и специальные файлы.

Ключи

ВЕРСИЯ УТИЛИТЫ FILE, ИМЕЮЩАЯСЯ В MAC OS X, ДОПУСКАЕТ ИСПОЛЬЗОВАНИЕ ДЛИННЫХ КЛЮЧЕЙ

Ключи утилиты file, перед которыми ставится двойной дефис (– –), работают как под Mac OS X, так и под Linux.

`--files-from=`***файл*** `-f` ***файл*** Получение имен проверяемых файлов из файла по имени ***файл***, а не из списка файлов в командной строке. Имена файлов в файле по имени ***файл*** должны быть перечислены по одному в каждой строке.

`--no-dereference -h` Для каждого файла, являющегося символьной ссылкой, вывод отчета о символьной ссылке, а не о файле, на который она указывает. При использовании этого ключа файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Эти действия осуществляются по умолчанию на тех системах, где переменная среды окружения `POSIXLY_CORRECT` (как обычно) не определена.

`--help` Вывод справки.

`--mime -I` Вывод строк типа MIME. (OS X)

`--mime -i` Вывод строк типа MIME. (Linux)

`--dereference -L` Для каждого файла, являющегося символьной ссылкой, вывод отчета о файле, на который указывает ссылка, а не о самой символьной ссылке. При использовании этого ключа файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Эти действия осуществляются по умолчанию на тех системах, где определена переменная среды окружения `POSIXLY_CORRECT`.

`--uncompress -z` (zip). Попытка классификации файлов, находящихся в сжатом файле.

Примечания

Утилита **file** может классифицировать более чем 5000 типов файлов. Наиболее распространенные типы файлов, используемые на Linux-системах, отображаются утилитой **file** в следующем виде:

```
archive
ascii text
c program text
commands text
core file
cpio archive
data
directory
ELF 32-bit LSB executable
empty
English text
executable
```

При попытке классифицировать файл утилита **file** использует максимум три теста: файловой системы, магического числа (системного кода) и языка. Тест файловой системы изучает значение, возвращаемое системным вызовом **stat( )**, чтобы определить, не является ли файл пустым или специальным. Тест магического числа ищет данные в конкретно определенных форматах ближе к началу файла. Тест языка определяет, если это необходимо, является ли файл текстовым, какая кодировка в нем используется и на каком языке он написан. Более подробное описание работы утилиты **file** дано на посвященной ей **man**-странице. Результаты работы утилиты **file** не всегда бывают соответствующими действительности.

Примеры

Далее приводится ряд примеров идентификации файлов:

```
/etc/Muttrc:            ASCII English text
/etc/Muttrc.d:          directory
/etc/adjtime:           ASCII text
/etc/aliases.db:        Berkeley DB (Hash, version 9, native byte-order)
/etc/at.deny:           writable, regular file, no read permission
/etc/bash_completion:   ASCII Pascal program text
/etc/blkid.tab.old:     Non-ISO extended-ASCII text, with CR, LF line terminators
/etc/brltty.conf:       UTF-8 Unicode C++ program text
/etc/chatscripts:       setgid directory
/etc/magic:             magic text file for file(1) cmd
/etc/motd:              symbolic link to `/var/run/motd'
/etc/qemu-ifup:         POSIX shell script text executable
/usr/bin/4xml:          a python script text executable
/usr/bin/Xorg:          ELF 64-bit LSB executable, x86-64, version 1 (SYSV), for GNU/
Linux
2.6.8, dynamically linked (uses shared libs), stripped
/usr/bin/debconf:       perl script text executable
/usr/bin/locate:        symbolic link to `/etc/alternatives/locate'
/usr/share/man/man7/hier.7.gz: gzip compressed data, was "hier.7", from Unix, last
modified: Thu Jan 31 03:06:00 2008, max compression
```

find

Поиск файлов, отвечающих определенным критериям.

*find [**список_каталогов**] [**ключ**] [**выражение**]*

Утилита find выбирает файлы, находящиеся в указанных иерархиях каталогов и отвечающих указанному критерию.

Аргументы

Список_каталогов указывает иерархию каталогов, в которых утилита find ведет поиск. Если ***список_каталогов*** не указан, find просматривает иерархию рабочего каталога.

Ключ управляет разыменованием символьных ссылок по мере нисходящего поиска утилиты find в иерархиях каталогов. По умолчанию find не занимается разыменованием символьных ссылок (работая с символьными ссылками, а не с файлами, на которые они указывают). При работе под управлением Mac OS X можно воспользоваться ключом `-x`, чтобы запретить утилите find вести поиск в каталогах файловой системы, не указанных в ***списке_каталогов***. При работе под управлением Linux ту же функцию выполняет критерий `-xdev`.

Выражение содержит критерий, рассматриваемый в разделе «Критерий». Утилита find проверяет каждый файл в каждом каталоге из ***списка_каталогов*** на соответствие критерию, описанному в ***выражении***. Если ***выражение*** не указано, используется его значение по умолчанию `-print`.

Пробел, разделяющий два критерия, является булевым оператором «И»: чтобы файл был выбран, он должен отвечать обоим критериям. Строка `-or` или `-o`, раз-

деляющая два критерия, является булевым оператором «ИЛИ»: чтобы файл был выбран, он должен отвечать либо одному, либо другому критерию (или обоим критериям).

Любой критерий можно инвертировать, поставив перед ним восклицательный знак. Если не использована группировка с помощью круглых скобок, утилита find вычисляет критерий слева направо.

Специальные символы внутри ***выражения*** должны быть процитированы, чтобы оболочка не смогла их интерпретировать и передала их утилите find. В число часто используемых с find специальных символов входят круглые, квадратные скобки, вопросительные знаки и звездочки.

Каждый элемент ***выражения*** является отдельным аргументом. Аргументы должны быть отделены друг от друга пробелами. Пробел должен ставиться с обеих сторон каждой круглой скобки, восклицательного знака, критерия и другого элемента.

Ключи

`-H` (с частичным разыменованием). Для каждого файла, являющегося символьной ссылкой, работа с файлом, на который указывает ссылка, а не с самой символьной ссылкой. Действие этого ключа распространяется на файлы, указанные в командной строке, оно не касается файлов, найденных при спуске по иерархии каталога. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L».

`-L` Для каждого файла, являющегося символьной ссылкой, работа с файлом, на который указывает ссылка, а не с самой символьной ссылкой. Действие этого ключа распространяется на все файлы. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример его использования приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L». (dereference)

`-P` (без разыменования). Для каждого файла, являющегося символьной ссылкой, работа с символьной ссылкой, а не с файлом, на который она указывает. Действие этого ключа распространяется на все файлы. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Эти действия осуществляются по умолчанию. Пример использования этого ключа приведен в разделе «chgrp», в подразделе «Примеры».

`-x` Принуждение find к отказу от поиска в каталогах файловой системы, кроме того (или тех), который указан в ***списке_каталогов***. При работе под управлением Linux используется критерий `-xdev`. (OS X)

Критерии

Внутри ***выражения*** можно использовать следующие критерии. В данном перечне элемент $\pm n$ представляет собой десятичное целое число, которое может рассматриваться как $+n$ (больше чем n), $-n$ (меньше чем n) или n (в точности n).

`-anewer` ***имя_файла***

(accessed newer — более свежее обращение). Вычисляемый файл отвечает этому критерию, если к нему обращались уже после обращения к файлу с именем ***имя_файла***.

`-atime ±n`

(access time — время обращения). Вычисляемый файл отвечает этому критерию, если к нему в последний раз обращались ±*n* дней назад. При использовании этого критерия утилита find изменяет время последнего обращения к просматриваемым каталогам.

`-depth`

Вычисляемый файл всегда отвечает этому активному критерию, который заставляет утилиту find предпринимать действие над записями в каталоге перед действиями в отношении самого каталога. Когда утилита find используется для отправки файлов утилите cpio, критерий `-depth` позволяет cpio сохранить время последнего изменения каталогов при восстановлении файлов (если предположить, что при вызове cpio использовался ключ `--preserve-modification-time`). См. подразделы «Комментарии» и «Примеры» в разделе, посвященном утилите cpio.

`-exec` *команда* `\;`

Вычисляемый файл отвечает этому критерию, если *команда* возвращает статус выхода 0 (нуль, [*true*]). Команда должна заканчиваться процитированной точкой с запятой. Пара фигурных скобок ({}) внутри *команды* представляет имя вычисляемого файла. Активный критерий `-exec` должен быть в конце группы остальных критериев, чтобы *команда* выполнялась только в том случае, когда файл будет отвечать всем предыдущим критериям. Дополнительные сведения даны в следующем разделе «Комментрии». Более эффективный способ получения такого же результата рассмотрен в разделе, посвященном утилите xargs.

`-group` *имя*

Вычисляемый файл отвечает этому критерию, если он связан с группой по имени *имя*. Вместо имени можно использовать числовой ID группы.

`-inum` *n*

Вычисляемый файл отвечает этому критерию, если номер его i-узла равен *n*.

`-links ±n`

Вычисляемый файл отвечает этому критерию, если на него имеется ±*n* ссылок.

`-mtime ±n`

(modify time — время последнего изменения). Вычисляемый файл отвечает этому критерию, если он последний раз изменялся ±*n* дней назад.

`-name` *имя_файла*

Вычисляемый файл отвечает этому критерию, если шаблон *имя_файла* соответствует его имени. *Имя_файла* может включать групповые символы (*, ? и []), но все они должны быть процитированы.

`-newer` *имя_файла*

Вычисляемый файл отвечает этому критерию, если он изменялся раньше файла по имени *имя_файла*.

`-nogroup`

Вычисляемый файл отвечает этому критерию, если он не принадлежит ни одной из групп, известных на локальной системе.

`-nouser`

Вычисляемый файл отвечает этому критерию, если он не принадлежит ни одному из пользователей, известных на локальной системе.

`-ok` ***команда*** `\;`

Этот активный критерий аналогичен критерию `-exec`, с той лишь разницей, что он выводит каждую выполняемую команду, взятую в угловые скобки в виде приглашения, и выполняет команду, только если получает со стандартного ввода ответ, который начинается с `y` или `Y`.

`-perm [±]`***режим***

Вычисляемый файл отвечает этому критерию, если к нему есть права доступа, заданные аргументом *режим*. Если перед режимом стоит знак минуса (`-`), права доступа к файлу должны включать в себя все биты *режима*. Например, если *режим* равен 644, то файл с правами доступа 755 будет отвечать этому критерию. Если перед *режимом* стоит знак плюса (`+`), права доступа должны включать по крайней мере один из битов *режима*. Если перед *режимом* не стоит плюс или минус, то режим доступа к файлу должен в точности соответствовать указанному *режиму*. Для *режима* можно использовать как символьную, так и восьмеричную нотацию (см. раздел, посвященный утилите chmod).

`-print`

Вычисляемый файл всегда отвечает этому активному критерию. Когда при вычислении ***выражения*** попадается данный критерий, утилита find выводит путевое имя вычисляемого файла. Если `-print` является единственным критерием ***выражения***, find выводит имена всех файлов, имеющихся в ***списке_файлов***. Если этот критерий появляется вместе с другими критериями, find выводит имя только в том случае, если файл отвечает условиям предыдущего критерия. Если в ***выражении*** отсутствует активный критерий, предполагается, что в нем используется критерий `-print`. (См. следующие разделы «Комментарии» и «Примечания».)

`-size ±`***n***`[c|k|M|G]`

Вычисляемый файл отвечает этому критерию, если его размер равен указанному в аргументе ±*n* количеству 512-байтных блоков. Для измерения длины в других единицах нужно после *n* поставить букву `c` для измерения в символах, `k` для измерения в килобайтах, `M` для измерения в мегабайтах и `G` для измерения в гигабайтах.

`-type` ***тип_файла***

Вычисляемый файл отвечает этому критерию, если он относится к типу, указанному в аргументе *тип_файла*. Для выбора *типа_файла* служит следующий список:

`b` – специальный блоковый файл;

`c` – специальный символьный файл;

`d` – файл каталога;

`f` – обычный файл;

`l` – символьная ссылка;

`p` – FIFO (именованный канал named pipe);

`s` – сокет

`-user` ***имя***

Вычисляемый файл отвечает этому критерию, если он принадлежит пользователю, чье имя пользователя указанно в аргументе ***имя***. Вместо имени можно воспользоваться ID пользователя.

`-xdev`

Вычисляемый файл всегда отвечает этому активному критерию, который не дает утилите find просматривать каталоги в файловой системе, не указанные в ***списке_каталогов***. Аналогичен критерию `-mount`. При работе под управлением Mac OS X используется ключ `-x`. (Linux)

Комментарии

Если предположить, что `x` и `y` — критерии, то следующая командная строка никогда не будет проверять, отвечает ли файл критерию `y`, если он не отвечает критерию `x`. Поскольку между критериями стоит пробел (булев оператор «И»), как только утилита find определит несоответствие критерию `x`, файл не сможет отвечать критерию, поэтому find не продолжает тестирование. Выражение можно прочитать как «(проверь подлинность того, что) файл отвечает критерию `x` *и* [пробел означает *и*] критерию `y`».

```
$ find dir x y
```

В следующей командной строке проверяется соответствие файла критерию `y`, если он не отвечает критерию `x`. Файл все же может ответить критерию, поэтому утилита find продолжает вычисление. Выражение можно прочитать как «(проверь подлинность того, что) файл отвечает критерию `x` или критерию `y`». Если файл отвечает критерию `x`, файл не вычисляет критерий `y`, поскольку в этом нет необходимости.

```
$ find dir x -or y
```

Активный критерий

Подобный «критерий» вместо участия в выборе файлов заставляет find выполнить действие. Это действие запускается, когда утилита find вычисляет один из таких ***активных критериев***. Поэтому выполнит find действие или нет, определяется не результатом вычисления активного критерия, а местом этого критерия в командной строке.

Активный критерий `-print` заставляет утилиту find вывести путевое имя проверяемого файла. В следующей командной строке выводятся имена всех файлов в каталоге dir (и во всех его подкаталогах), независимо от того, отвечают они или нет критерию `x`:

```
$ find dir -print x
```

В следующей командной строке выводятся имена лишь тех файлов в каталоге dir, которые отвечают критерию `x`:

```
$ find dir x -print
```

Применение `-print` после критерия проверки является активным критерием, выполняемым по умолчанию. В следующей командной строке генерируется такой же вывод, как и в предыдущей:

```
$ find dir x
```

Примечания

Чтобы понятнее выразить условия, между критериями можно поставить оператор -a. Этот оператор, точно так же, как и пробел, является булевым оператором «И».

Вместо утилиты cpio можно вопользоваться утилитой pax (см. соответствующий раздел).

Примеры

Самая простая команда find не использует аргументов и выводит список файлов, имеющихся в рабочем каталоге и всех его подкаталогах:

```
$ find
...
```

В следующей команде ведется поиск файлов в рабочем каталоге и его подкаталогов, имена которых начинаются на «a». Рабочий каталог в этой команде обозначен точкой. Чтобы оболочка не интерпретировала символы a* в качестве нестрогой ссылки на имена файлов, они взяты в одинарные кавычки.

```
$ find . -name 'a*'
```

В предыдущей команде скрытно используется критерий -print. Если опустить аргумент ***список_каталогов***, утилита find ведет поиск в рабочем каталоге. Следующая команда выполняет ту же функцию, что и предыдущая, но без явного указания рабочего каталога:

```
$ find -name 'a*'
```

В следующей команде список выбранных имен файлов отправляется утилите cpio, которая записывает его на устройство, смонтированное в каталоге /dev/sde1. Первая часть командной строки заканчивается символом канала, поэтому оболочка ожидает следования еще одной команды и выводит перед получением оставшейся части командной строки вторичное приглашение (>). Эту команду find можно прочитать как «нужно найти в корневом каталоге и всех подкаталогах (/), обычные файлы (-type f), измененные за последний день (-mtime -1), за исключением тех файлов, у имен которых есть суффикс .o (! -name '*.o')». (Суффикс .o относится к объектному файлу, который обычно не нужно сохранять, потому что он может быть воссоздан из соответствующего исходного файла.)

```
$ find / -type f -mtime -1 ! -name '*.o' -print |
> cpio -oB > /dev/sde1
```

Следующая команда ведет поиск, вывод имен файлов и удаление файлов с именами core или junk из рабочего каталога и всех его подкаталогов:

```
$ find . \( -name core -o -name junk \) -print -exec rm {} \;
...
```

Круглые скобки и точка с запятой, следующая за критерием -exec, процитированы, поэтому оболочка не рассматривает их в качестве специальных символов. Круглые скобки отделены от других элементов командной строки пробелами. Эту команду find можно прочитать как «нужно найти в рабочем каталоге и подкаталогах (.) файлы по имени core (-name core) *или* (-o) junk (-name junk) [если файл отвечает этим критериям, продолжить] *и* (ПРОБЕЛ) вывести имя файла (-print) *и* (ПРОБЕЛ) удалить файл (-exec rm { })».

В следующем сценарии оболочки утилита find используется совместно с утилитой grep для идентификации файлов, которые содержат определенную строку. Этот сценарий позволяет искать файл по запомненному содержимому при забытом имени файла. Сценарий finder определяет местонахождение в рабочем каталоге и его подкаталогах тех файлов, которые содержат символьную строку, указанную в командной строке. Критерий `-type f` заставляет утилиту find передать утилите grep имена только обычных файлов, а не каталогов.

```
$ cat finder
find . -type f -exec grep -l "$1" {} \;
$ finder «Executive Meeting"
 ./january/memo.0102
 ./april/memo.0415
```

При вызове со строкой `Executive Meeting` сценарий finder выводит местоположение двух файлов, содержащих эту строку: ./january/memo.0102 и ./april/memo.0415. Точка (.) в путевом имени представляет рабочий каталог, а january и april являются подкаталогами рабочего каталога. Утилита grep с ключом `--recursive` выполняет ту же функцию, что и сценарий finder.

Следующая команда ищет в двух пользовательских каталогах файлы, размер которых превышает 100 блоков (`-size +100`) и обращение к которым происходило не ранее пяти дней назад, то есть к файлам за последние пять дней не обращались (`-atime +5`). Затем эта команда find запрашивает, нужно ли удалить файл (`-ok rm {}`). На каждый запрос нужно просто ответить `y` (*да*) или `n` (*нет*). Команда rm работает, только если у вас имеются права на запись каталога и на его выполнение.

```
$ find /home/max /home/hls -size +100 -atime +5 -ok rm {} \;
< rm … /home/max/notes >? y
< rm … /home/max/letter >? n
…
```

В следующем примере /home/max/memos является символьной ссылкой на каталог пользователя Sam по имени /home/sam/memos. При использовании с утилитой find ключа `-follow` осуществляется переход по символьной ссылке, и поиск ведется в указанном каталоге.

```
$ ls -l /home/max
lrwxrwxrwx  1 max   pubs    17 Aug 19 17:07 memos -> /home/sam/memos
-rw-r--r--  1 max   pubs  5119 Aug 19 17:08 report

$ find /home/max -print
/home/max
/home/max/memos
/home/max/report
/home/max/.profile

$ find /home/max -follow -print
/home/max
/home/max/memos
/home/max/memos/memo.817
/home/max/memos/memo.710
/home/max/report
/home/max/.profile
```

finger

Вывод сведений о пользователях.

finger [ключи] [список_пользователей]

Утилита finger выводит пользовательские имена наряду с полными именами пользователей, номерами терминальных устройств, временем их входа в систему и другой информацией. Ключи управляют количеством выводимой утилитой finger информации, а ***список_пользователей*** определяет пользователей, информацию о которых выводит finger. Утилита finger может извлекать информацию как из локальной, так и из удаленных систем.

Аргументы

Без использования аргументов утилита finger предоставляет краткий (-s) отчет о пользователях, вошедших в локальную систему. При указании ***списка_пользователей*** finger предоставляет развернутый (-l) отчет по каждому пользователю из этого списка. Имена в ***списке_пользователей*** не различаются по регистру букв.

Если имя включает знак at (@), утилита finger интерпретирует имя, следующее за символом @ как имя удаленного хоста, предназначенного для подключения по сети. Если имя пользователя появляется перед символом @, finger предоставляет информацию об этом пользователе, имеющуюся на удаленной системе.

Ключи

-l (long — длинный). Вывод подробной информации (этот режим вывода используется по умолчанию в случае указания ***списка_пользователей***).

-m (match — соответствующий). Если указан ***список_пользователей***, вывод записей, касающихся только тех пользователей, чьи пользовательские имена соответствуют одному из имен в ***списке_пользователей***. Без этого ключа имена в ***списке_пользователей*** соответствуют как пользовательским, так и полным именам.

-p (без plan, project или pgpkey). Отмена для пользователей вывода содержимого файлов .plan, .project и .pgpkey. Поскольку в этих файлах могут содержаться эскейп-последовательности, использующие обратный слэш, которые способны изменять поведение экрана, они могут быть нежелательны для просмотра. Обычно содержимое этих файлов, если они присутствуют в личных каталогах пользователей, выводится в подробных списках.

-s (short — краткий). Предоставление краткого отчета для каждого пользователя (этот режим вывода используется по умолчанию в том случае, если не указан ***список_пользователей***).

Комментарии

В подробный отчет, предоставляемый утилитой finger, включается пользовательское имя, полное имя, место, где находится личный каталог и оболочка входа в систему, плюс информация о времени последнего входа пользователя в систему и сколько

времени прошло с тех пор, как он в последний раз осуществлял набор на клавиатуре и читал свою электронную почту. После извлечения этой информации из системных файлов утилита finger выводит содержимое файлов ~/.plan, ~/.project и ~/.pgpkey, находящихся в личных каталогах пользователей. Создание и поддержка этих файлов, которые обычно предоставляют дополнительные сведения о пользователе (например, телефонные номера, почтовый адрес, личный план, интересы и PGP-ключ), — личное дело каждого пользователя.

Краткий отчет, создаваемый утилитой finger, похож на отчет, предоставляемый утилитой w, и включает в себя пользовательское имя, полное имя пользователя, номер устройства пользовательского терминала, количество времени, прошедшее с момента ввода им последнего символа с клавиатуры терминала, время входа пользователя в систему и местонахождение пользовательского терминала. Если пользователь вошел в систему по сети, утилита finger выводит имя удаленной системы.

Примечания

Не все дистрибутивы Linux предусматривают установку утилиты finger по умолчанию.

При указании сетевого адреса утилита finger запрашивает стандартную сетевую службу, запущенную на удаленной системе. Хотя такая служба предоставляется большинством Linux-систем, некоторые администраторы предпочитают ее не запускать (чтобы минимизировать загруженность своих систем, сократить возможные риски безопасности или упростить себе обслуживание системы). Если попытаться навести справки о ком-нибудь на подобном сайте, в результате может быть возвращено сообщение об ошибке или же запрос вообще может быть оставлен без ответа. Объем информации, которой готова поделиться с локальной системой удаленная система, и формат этой информации определяются на удаленной системе. В результате этого отчет, отображаемый для каждой конкретной системы, может отличаться от приводимых здесь примеров. См. также раздел «finger: Вывод списка пользователей, вошедших в систему» на с. 88.

Файл по имени ~/.nofinger заставляет утилиту finger отрицать существование того человека, в чьем личном каталоге он находится. Чтобы эта уловка работала, запрос утилиты finger должен исходить из системы, отличающейся от локального хоста, и демон fingerd должен иметь возможность видеть файл .nofinger (обычно личный каталог должен иметь установленный бит права выполнения для всех остальных пользователей).

Примеры

В первом примере выводится информация о пользователях, зарегистрировавших свой вход на локальной системе:

```
$ finger
Login    Name              Tty    Idle  Login  Time  Office  Office Phone
max      Max Wild          tty1   13:29 Jun 25 21:03
hls      Helen Simpson     *pts/1 13:29 Jun 25 21:02 (:0)
sam      Sam the Great     pts/2        Jun 26 07:47 (bravo.example.com)
```

Звездочка (*) впереди имени терминала пользователя Helen (TTY), показывает, что этот пользователь заблокировался от отправки сообщений непосредственно на ее тер-

минал (см. раздел «mesg: Отказ в приеме или прием сообщений» на с. 91). Подробный отчет выводит сведения о выключении строковых сообщений для тех пользователей, которые отключили сообщения.

В следующих двух примерах утилита finger вынуждена обратиться по сети за информацией к удаленной системе по имени kudos:

```
$ finger @kudos
[kudos]
Login    Name              Tty    Idle  Login Time   Office  Office Phone
max      Max Wild          tty1   23:15 Jun 25 11:22
roy      Roy Wong          pts/2        Jun 25 11:22

$ finger watson@kudos
[kudos]
Login: max                              Name: Max Wild
Directory: /home/max                    Shell: /bin/zsh
On since Sat Jun 25 11:22 (PDT) on tty1,  idle 23:22
Last login Sun Jun 26 06:20 (PDT) on ttyp2 from speedy
Mail last read Thu Jun 23 08:10 2005 (PDT)
Plan:
For appointments contact Sam the Great, x1963.
```

fmt

Простейшее форматирование текста.

fmt [ключ] [список_файлов]

Утилита fmt осуществляет простое форматирование текста путем придания каждой непустой строке примерно одинаковой длины.

Аргументы

Утилита fmt читает файлы из ***списка_файлов*** и отправляет отформатированную версию их содержимого на стандартный вывод. Если имя файла не указано или вместо него указан дефис (-), fmt читает данные со стандартного ввода.

Ключи

Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--split-only`
`-s` Разбиение длинных строк без дополнения коротких. (Linux)

`-s` Замена нескольких смежных пробелов одним. (OS X)

`--tagged-paragraph`
`-t` Отступы во всех, кроме первой строки каждого абзаца. (Linux)

`-t` *n* Указание с помощью *n* числа пробелов, эквивалентных переходу по табуляции. По умолчанию это число равно восьми. (OS X)

`--uniform-spacing` `-u` Изменение отформатированного вывода таким образом, чтобы между словами был один пробел, а между предложениями — два. (Linux)

`--width=n` `-w n` Изменение длины строки вывода до *n* символов. Без этого ключа fmt придерживается примерной длины строки в 75 символов. Этот ключ можно также указать как *-n*.

Примечания

Утилита fmt работает за счет перемещения разделителей строк. Отступы строк и промежутки между словами остаются нетронутыми.

Эту утилиту можно использовать для форматирования текста при использовании такого редактора, как, к примеру, vim. Например, можно отформатировать абзац с помощью редактора vim в Командном режиме, если установить курсор в начало абзаца и ввести команду `!}fmt -60`. Эта команда заменит абзац тем выводом, который будет создан после пропуска этого абзаца через утилиту fmt, которой указана ширина строки 60 символов. Если нужно отменить форматирование, следует тут же ввести команду `u`.

Примеры

В следующем примере показано, как fmt пытается придать всем строкам одинаковую длину. Ключ `-w 50` устанавливает длину получаемой строки равной 50 символам.

```
$ cat memo
One factor that is important to remember while administering the dietary
intake of Charcharodon carcharias is that there is, at least from
the point of view of the subject,
very little
differentiating the prepared morsels being proffered from your digits.

In other words, don't feed the sharks!

$ fmt -w 50 memo
One factor that is important to remember while
administering the dietary intake of Charcharodon
carcharias is that there is, at least from the
point of view of the subject, very little
differentiating the prepared morsels being
proffered from your digits.

In other words, don't feed the sharks!
```

В следующем примере демонстрируется действие ключа `--split-only`. Длинные строки разбиваются так, чтобы они были не длиннее 50 символов. Этот ключ не позволяет fmt дополнять короткие строки.

```
$ fmt -w 50 --split-only memo
One factor that is important to remember while
administering the dietary
intake of Charcharodon carcharias is that there
is, at least from
the point of view of the subject,
```

```
very little
differentiating the prepared morsels being
proffered from your digits.

In other words, don't feed the sharks!
```

fsck

Проверка и восстановление файловой системы.

fsck [ключи] [список_файловых_систем]

Утилита fsck проверяет целостность файловой системы и выводит отчет о найденных проблемах, а также дополнительно занимается их устранением. Она предоставляет внешний интерфейс для средств проверки файловых систем, каждое из которых имеет свою специфику применительно к конкретному типу файловой системы. Утилита fsck доступна только при работе под управлением Linux, а в OS X используется утилита diskutil. (Linux)

Аргументы

Без ключа -A и без указания ***списка_файловых_систем*** утилита fsck поочередно (последовательно) проверяет файловые системы, перечисленные в файле /etc/fstab. При вызове с ключом -A и без указания ***списка_файловых_систем*** утилита fsck проверяет, если это возможно, в параллельном режиме все файловые системы, перечисленные в файле /etc/fstab. Параллельная проверка файловых систем рассмотрена в описании ключа -s.

Список_файловых_систем указывает те файловые системы, которые будут подвергаться проверке. В нем может быть указано либо имя устройства, содержащего файловую систему (например, /dev/hda2), либо, если файловая система фигурирует в файле /etc/fstab, каталог, к которому она примонтирована (например, /usr). ***Список_файловых_систем*** может также указывать метку для файловой системы из файла /etc/fstab (например, LABEL=home).

Ключи

При запуске утилиты fsck можно указать как глобальные ключи, так и ключи, специфичные для типа файловой системы, проверяемой fsck (например, ext2, ext3, msdos, reiserfs). Глобальные ключи должны предшествовать ключам, специфичным для типа файловой системы.

Глобальные ключи

-A (all — все). По возможности параллельная обработка всех файловых систем, перечисленных в файле /etc/fstab. Параллельная проверка файловых систем рассмотрена в описании ключа -s. При использовании этого ключа ***список_файловых_систем*** указывать не нужно; типы проверяемых файловых систем можно указать с помощью ключа -t. Чтобы утилита fsck не пыталась обрабатывать файловые системы

в параллельном интерактивном режиме (в случае чего было бы невозможно отвечать на ее многочисленные приглашения), этот ключ нужно использовать вместе с ключами -a, -p или -n.

-N (по — нет). Предполагает отрицательный ответ на любые вопросы, поднимаемые в процессе обработки файловой системы. Этот ключ вызывает выдачу сообщений, которые обычно можно увидеть, но заставляет fsck не предпринимать каких-либо действий.

-R (root-skip — пропуск корневой системы). При использовании с ключом -A, препятствует проверке корневой файловой системы. Этот ключ применяется при загрузке системы, поскольку для корневой файловой системы может повыситься объем обращений по чтению-записи.

-s (serial — последовательная обработка). Заставляет fsck обрабатывать файловые системы поочередно. Без этого ключа fsck параллельно обрабатывает несколько файловых систем, расположенных на отдельных физических дисковых приводах. Параллельная обработка позволяет fsck ускорить работу с несколькими файловыми системами. Этот ключ следует применять в том случае, если нужна интерактивная обработка файловых систем. Чтобы выключить интерактивную обработку, следует пользоваться ключами -a, -p или -N (или -n, на некоторых файловых системах).

-T (title — заголовок). Заставляет fsck не выводить ее заголовок.

-t *fstype* (filesystem type — тип файловой системы). Список с запятой в качестве разделителя, в котором указываются типы (тип) обрабатываемых файловых систем. При использовании ключа -A fsck обрабатывает все файловые системы, перечисленные в файле /etc/fstab и имеющие тип *fstype*. Наиболее распространенными типами файловых систем являются ext2, ext3, ext4, msdos и reiserfs. Проверка удаленных файловых систем NFS обычно не проводится.

-V (verbose — подробно). Вывод более подробной информации, включая команды, связанные со спецификой файловой системы.

Ключи, связанные со спецификой файловой системы

Следующая команда выводит список утилит проверки, доступных на локальной системе. Фалы с одинаковыми номерами i-узлов связаны друг с другом (см. раздел «Утилита ls и i-узлы» на с. 129).

```
$ ls -i /sbin/*fsck*
63801 /sbin/dosfsck  63856 /sbin/fsck.cramfs  63801 /sbin/fsck.msdos
63763 /sbin/e2fsck   63763 /sbin/fsck.ext2    63801 /sbin/fsck.vfat
63780 /sbin/fsck     63763 /sbin/fsck.ext3
```

Чтобы определить, какие ключи используются утилитой, нужно просмотреть man-страницу или ввести путевое имя утилиты проверки файловой системы:

```
$ /sbin/fsck.ext3
Usage: /sbin/fsck.ext3 [-panyrcdfvstDFSV] [-b superblock] [-B blocksize]
                [-I inode_buffer_blocks] [-P process_inode_size]
                [-l|-L bad_blocks_file] [-C fd] [-j ext-journal]
```

```
                [-E extended-options] device
Emergency help:
 -p        Automatic repair (no questions)   Автовосстановление (без вопросов)
 -n        Make no changes to the filesystem Отказ от изменений файловой системы
 -y        Assume "yes" to all questions     Ответ «yes» на все вопросы
 -c        Check for bad blocks and add them to the badblock list
             Проверка на сбойные блоки и добавление их в список сбойных блоков
 -f        Force checking even if filesystem is marked clean
         Принудительная проверка, даже если файловая система помечена как очищенная
...
```

Следующие ключи применимы ко многим типам файловых систем, включая ext2 и ext3:

`-a` (automatic — автоматически). Аналог ключа `-p`, сохраненный в целях обратной совместимости.

`-f` (force — принудительно). Заставляет fsck проверять даже уже очищенную файловую систему. К таким файловым системам относятся только что прошедшие успешную проверку с помощью утилиты fsck или успешно размонтированные и до сих пор не смонтированные. Очищенные файловые системы пропускаются утилитой fsck, что существенно ускоряет загрузку системы при нормальных условиях. Сведения о настройке периодической, автоматической проверки файловых систем ext2, ext3 и ext4 даны в описании утилиты tune2fs.

`-n` (no — нет). То же самое, что и глобальный ключ `-N`. Работает не со всеми файловыми системами.

`-p` (preen — исправить). Попытка устранить все мелкие несоответствия, обнаруженные в процессе обработки файловой системы. Если некоторые проблемы не устранены, fsck прекращает работу с ненулевым статусом выхода. Этот ключ вызывает запуск fsck в пакетном режиме, следовательно, утилита не запрашивает, нужно ли исправлять обнаруженную ею проблему. При проверке файловых систем в процессе загрузки Linux ключ `-p` чаще всего используется с ключом `-A`.

`-r` (interactive — интерактивный). Запрос на исправление или игнорирование каждой обнаруженной проблемы. Для многих типов файловых систем этот режим работает по умолчанию. Этот ключ доступен не во всех файловых системах.

`-y` (yes — да). Положительный ответ на все вопросы, задаваемые fsck при обработке файловой системы. Это ключ следует использовать с оглядкой, поскольку он предоставляет утилите fsck свободу делать все, что ей представляется наилучшим при очистке файловой системы от дефектов.

Примечания

Утилиты fsck и fsck_hfs не рекомендуется использовать при работе под управлением Mac OS X версии 10.5 и выше. Компания Apple предлагает воспользоваться вместо них утилитой diskutil (см. соответствующий раздел).

Утилиту fsck можно запустить с компакт-диска аварийного восстановления.

Когда файловая система не имеет несоответствий, fsck выводит отчет, имеющий следующий вид:

```
# fsck -f /dev/sdb1
fsck 1.40.8 (13-Mar-2008)
e2fsck 1.40.8 (13-Mar-2008)
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
/dev/sdb1: 710/4153408 files (10.1% non-contiguous), 455813/8303589 blocks
```

Интерактивный режим

Утилиту fsck можно запускать либо в интерактивном, либо в пакетном режиме. Пока не используется один из ключей -a, -p, -y или -n, для многих файловых систем fsck запускается в интерактивном режиме. Если утилита fsck обнаруживает проблему в файловой системе в интерактивном режиме, она рапортует о проблеме и дает возможность выбрать, устранять ее или игнорировать. При устранении проблемы может быть потеряна часть данных, но чаще всего это наиболее разумный выбор.

Хотя восстановление тех файлов, которые утилита fsck предлагает удалить, с технической точки зрения вполне возможно, такое восстановление практикуется довольно редко. Наилучшая гарантия от существенной потери данных — частое создание резервных копий.

Порядок проверки

Утилита fsck просматривает шестой столбец в файле /etc/fstab, чтобы определить, нужно ли проверять файловые системы и в какой последовательности это делать. Нуль (0) в этой позиции показывает, что файловую систему проверять не нужно. Единица (1) показывает, что ее нужно проверить в первую очередь; этот статус обычно резервируется за корневой файловой системой. Двойка (2) показывает, что файловая система должна проверяться после той, которая помечена единицей.

fsck является внешним интерфейсом

Так же как и утилита mkfs (см. соответствующий раздел), fsck является внешним интерфейсом, вызывающим другие утилиты для работы с различными типами файловых систем. Например, fsck вызывает утилиту e2fsck для проверки широко распространенных файловых систем ext2 и ext3. Дополнительные сведения по утилите e2fsck даны на посвященной ей man-странице. Другие утилиты, вызываемые fsck, обычно называются fsck.*тип*, где *тип* — это тип файловой системы. Путем такого разделения разработчики файловых систем могут предоставить программы для их проверки, не оказывая влияния на разработчиков других файловых систем или на порядок использования системным администратором утилиты fsck.

Время загрузки

Утилита fsck запускается для файловых систем, которые не смонтированы или смонтированы в режиме только для чтения. При загрузке Linux корневая файловая система сначала монтируется в режиме только для чтения, что позволяет обработать ее с помощью утилиты fsck. Если fsck не обнаружит проблем в корневой файловой системе, то она проходит перемонтировку (с помощью ключа

remount и утилиты mount) на использование в режиме чтение-запись, и fsck обычно запускается с ключами -A, -R и -p.

lost+found Когда будет обнаружен файл, утративший связь со своим именем файла, fsck спрашивает, нужно ли возобновить связь. Если будет выбрано возобновление связи, файл помещается в каталог по имени lost+found, находящийся в корневом каталоге той файловой системы, в которой был найден файл. Файлу с возобновленной связью в качестве имени присваивается номер его i-узла. Чтобы fsck восстанавливала файлы подобным образом, каталог lost+found должен присутствовать в корневом каталоге каждой файловой системы. Например, если система использует файловые системы /, /usr и /home, требуется наличие трех каталогов lost+found: /lost+found, /usr/lost+found и /home/lost+found. Каждый каталог lost+found должен иметь неиспользованные записи, в которых fsck сможет хранить номера i-узлов для потерявших свои связи файлов. При создании файловых систем ext2/ext3/ext4 утилита mkfs (см. соответствующий раздел) создает каталог lost+found с требуемыми неиспользованными записями. Кроме этого, если нужно, для создания этого каталога в файловых системах ext2/ext3/ext4 можно использовать утилиту mklost+found. Для других файловых систем можно создать неиспользованные записи путем добавления множества файлов к каталогу с их последующим удалением. Для создания 500 записей в каталоге lost+found попробуйте воспользоваться утилитой touch (см. соответствующий раздел) для создания 500 записей в каталоге lost+found, а затем используйте утилиту rm для удаления этих файлов.

Сообщения

В табл. V.15 показан перечень наиболее распространенных сообщений. Чаще всего утилита fsck предлагает наиболее логичный способ устранения проблемы, возникшей в файловой структуре. При отсутствии информации, подсказывающей другой ответ, нужно отвечать на приглашение утвердительно. Данные, утраченные в результате этого процесса, следует восстанавливать с помощью лент или дисков резервного копирования.

Таблица V.15. Наиболее распространенные сообщения утилиты fsck

Фаза (сообщение)	Что проверяет fsck
Фаза 1 — Checking inodes, blocks, and sizes (Проверка i-узлов, блоков и размеров)	Проверка информации об i-узлах
Фаза 2 — Checking directory structure (Проверка структуры каталогов)	Поиск каталогов, указывающих на сбойные i-узлы, найденные в фазе 1
Фаза 3 — Checking directory connectivity (Проверка связанности каталогов)	Поиск каталогов, на которые не имеется ссылок, а также установка отсутствия или заполнения каталога lost+found
Фаза 4 — Checking reference counts (Проверка количества ссылок)	Проверка файлов, на которые нет ссылок, а также отсутствия или заполнения каталога lost+found, неверного количества ссылок, сбойных блоков, продублированных блоков и неверного количества i-узлов

Фаза (сообщение)	Что проверяет fsck
Фаза 5 — Checking group summary information (Проверка итоговой групповой информации)	Проверка исправности списка свободных блоков и других структур файловой системы. Если обнаружатся проблемы со списком свободных блоков, запускается фаза 6
Фаза 6 — Salvage free list (Исправление списка свободных блоков)	Если в фазе 5 обнаружатся проблемы со списком свободных блоков, они устраняются при выполнении фазы 6

Наведение порядка

После ремонта файловой системы утилита fsck сообщает о ее состоянии, выводя следующее сообщение:

`*****File System Was Modified*****` *Файловая система была изменена*

В отношении файловых систем ext2/ext3/ext4, fsck выводит после их проверки следующее сообщение:

файловая система: используется/максимальное количество файлов (процент несмежных файлов), используется/максимальное количество блоков

Это сообщение информирует о количестве используемых файлов и блоков на диске, а также о том количестве файлов и блоков, которые могут содержаться в файловой системе. Процент несмежных файлов сообщает о степени фрагментированности диска.

ftp

Передача файлов по сети.

ftp [ключи] [удаленная_система]

Утилита ftp представляет собой пользовательский интерфейс к стандартному протоколу передачи файлов — File Transfer Protocol (FTP), по которому файлы передаются между системами, способными связываться по сети. Для установки FTP-соединения нужно иметь доступ к учетной записи (персональный, гостевой или анонимный) на удаленной системе.

> **ИСПОЛЬЗУЙТЕ FTP ТОЛЬКО ДЛЯ ЗАГРУЗКИ ОТКРЫТОЙ ИНФОРМАЦИИ**
>
> FTP не относится к безопасным протоколам. Утилита ftp отправляет ваш пароль по сети открытым текстом, что не считается безопасной процедурой. Если на сервере запущена оболочка OpenSSH, в качестве безопасной замены ftp можно воспользоваться sftp. Для многих FTP-функций, кроме загрузки информации анонимными пользователями, можно также воспользоваться утилитой scp (см. соответствующий раздел). Поскольку scp использует закодированную связь, пользовательские пароли и данные не могут быть перехвачены.

Аргументы

Удаленная_система — это имя или сетевой адрес сервера, на котором запущен FTP-демон (например, ftpd, vsftpd или sshd) и с которым нужно осуществить обмен файлами.

Ключи

`-i` (interactive — интерактивный режим). Выключение приглашений в процессе передачи с помощью команд mget и mput. См. также описание команды prompt.

`-n` (no automatic login — без автоматического входа в систему) Отключение автоматических входов в систему.

`-p` (passive mode — пассивный режим). Запуск ftp в пассивном режиме. См. описание команды passive.

`-v` (verbose — подробно). Более подробное освещение работы ftp. Вывод ответов ***удаленной_системы*** и отчетов о времени и скоростях передачи.

Комментарии

Утилита ftp работает в интерактивном режиме. После запуска она выводит приглашение на ввод команд для установки параметров и передачи файлов. В ответ на приглашение `ftp>` можно воспользоваться множеством команд, наиболее распространенные из которых перечислены ниже.

![*команда*] Переход в оболочку (ее вызов) на локальной системе. После завершения использования локальной оболочки для возвращения в ftp нужно воспользоваться комбинацией CONTROL+D или командой exit. Чтобы выполнить только одну ***команду***, ее нужно указать за восклицательным знаком как только ***команда*** завершит свою работу, ftp вернется к приглашению `ftp>`. Поскольку оболочка, вызываемая ftp вместе с этой командой, является дочерней по отношению к той оболочке, которая выполняет ftp, никакие изменения, внесенные в эту оболочку при возвращении в ftp, сохранены не будут. В частности, когда нужно скопировать файлы в локальный каталог, отличный от того каталога, из которого была запущена ftp, нужно воспользоваться ftp-командой lcd, чтобы изменить локальный рабочий каталог: ввод команды cd в вызванной оболочке не приведет к желаемым изменениям. См. пример в разделе «Примеры» в подразделе «lcd (локальная cd)».

ascii Установка типа передаваемых файлов в ASCII. Эта команда позволяет передавать текстовые файлы из систем, которые завершают строки комбинацией возврат каретки — перевод строки, и автоматически избавляться от возврата каретки. Такая передача применяется, когда удаленный компьютер является машиной, работающей под управлением DOS или MS Windows. Чтобы команда ascii работала, команда cr должна быть дана с аргументом `ON`.

binary Установка типа передаваемых файлов в двоичный. Эта команда позволяет правильно передавать файлы, содержащие не-ASCII (не выводимые на печать) символы. Она также работает для ASCII-файлов, не требующих изменения окончаний строк.

bye Закрытие подключения к удаленной системе и прекращение работы ftp. Аналогична команде quit.

cd *удаленный_каталог* Смена рабочего каталога на удаленной системе на каталог *удаленный_каталог*.

close Закрытие подключения к удаленной системе без выхода из ftp.

cr (carriage return — возврат каретки). Переключение удаления символа возврата каретки при извлечении файлов в ASCII-режиме. См. описание команды ascii.

dir [*каталог* [*файл*]] Вывод листинга каталога по имени *каталог* из удаленной системы. Если *каталог* не указан, выводится листинг рабочего каталога. Когда указан *файл*, листинг сохраняется на локальной системе в файле по имени *файл*.

get *удаленный_файл* [*локальный_файл* Копирование *удаленного_файла* на локальную систему под именем *локальный_файл*. Без указания *локального_файла* ftp использует в качестве имени на локальной системе имя *удаленного_файла*. Имена *удаленный_файл* и *локальный_файл* могут быть путевыми именами.

glob Включение и выключение расширения имени файла для команд mget и mput и вывод текущего состояния (`Globbing on` или `Globbing off`).

help Вывод списка команд, распознаваемых утилитой ftp на локальной системе.

lcd [*локальный_каталог*] (local change directory — локальная смена каталога). Смена рабочего каталога на локальной системе; делает рабочим *локальный_каталог*. Без аргумента эта команда меняет рабочий каталог на локальной системе, делая рабочим личный каталог (точно так же, как и команда cd, вызванная без аргумента).

ls [*каталог* [*файл*]] Аналогична команде dir, но выводит более краткий листинг для некоторых удаленных систем.

mget *список_удаленных_файлов* (multiple get — множественное получение). В отличие от команды get позволяет получать с удаленной системы сразу несколько файлов. Можно задать имена удаленных файлов буквально или воспользоваться групповыми символами (см. команду glob). См. также описание команды prompt.

mput *список_локальных_файлов* (multiple put — множественная отдача). Команда mput позволяет копировать сразу несколько файлов с локальной системы на удаленную. Можно задать имена локальных файлов буквально или воспользоваться групповыми символами (см. команду glob). См. также описание команды prompt.

open Интерактивное указание имени удаленной системы. Эта команда применяется в том случае, если удаленная система не указана в командной строке или если попытка подключиться к системе закончилась неудачно.

passive Переключение между активным (`PORT` — по умолчанию) и пассивным (`PASV`) режимами передачи данных и вывод режима передачи. См. подраздел «Сравнение пассивного и активного подключения» в разделе «Примечания» на с. 718.

prompt При использовании mget или mput для получения или отправки нескольких файлов ftp запрашивает подтверждение (по умолчанию) перед передачей каждого файла. Эта команда включает и выключает это поведение и выводит текущее состояние (`Interactive mode off` или `Interactive mode on`).

put *локальный_файл* [*удаленный_файл*] Копирование *локального_файла* на удаленную систему под именем *удаленный_файл*. Без указания *удаленного_файла* ftp использует в качестве имени файла на удаленной системе *локальный_файл*. *Удаленный_файл* и *локальный_файл* могут быть путевыми именами.

pwd Принуждение ftp к выводу путевого имени каталога удаленной системы. Для вывода имени локального рабочего каталога используется команда !pwd.

quit Закрытие подключения к удаленной системе и завершение работы ftp. Аналогична команде bye.

reget *удаленный_файл* Попытка возобновления прерванной передачи. Эта команда аналогична команде get, но вместо перезаписи существующего локального файла ftp добавляет к нему новые данные. Команда reget поддерживается не всеми серверами.

user [*имя_пользователя*] Если утилита ftp не осуществляет для вас автоматический вход в систему, то имя своей учетной записи можно указать как *имя_пользователя*. Если опустить *имя_пользователя*, ftp выведет приглашение на его ввод.

Примечания

Система Linux или Mac OS X с запущенной утилитой ftp может вести обмен файлами с любой из многих операционных систем, поддерживающих FTP-протокол. Многие сайты предлагают на FTP-сервере архивы свободно распространяемой информации, хотя многие из таких сайтов являются простыми альтернативами более легким по доступу веб-сайтам (например, ftp://ftp.ibiblio.org/pub/Linux и http://www.ibiblio.org/pub/Linux). Многие браузеры способны подключаться к FTP-серверам и загружать с них файлы.

Утилита ftp не выстраивает предположений насчет названия файловой системы или ее структуры, поскольку ftp можно использовать для обмена файлами с системами, не принадлежащими семейству UNIX/Linux (чьи соглашения об именах файлов могут отличаться от принятых в этих системах).

Анонимный FTP Многие системы, чаще всего те, с которых можно загрузить свободно распространяемое программное обеспечение, позволяют осуществлять анонимный вход. Большинство систем, поддерживающих анонимный вход, используют имя ftp как простой в написании и быстрый для набора синоним для anonymous. Доступ анонимному пользователю к файловой системе обычно ограничивается той ее частью, которая выделена для хранения файлов, совместно используемых с удаленными пользователями. При входе в систему в качестве анонимного пользователя сервер предлагает вам ввести пароль. Хотя может быть принят любой пароль, по соглашению от вас ожидается предоставление адреса вашей электронной почты. Многие системы, допускающие анонимный доступ, хранят интересные файлы в каталоге pub.

Сравнение пассивного и активного подключения Клиент может запросить FTP-сервер установить для передачи данных либо PASV (пассивное, используется по умолчанию), либо PORT (активное) подключение. Некоторые серверы ограничены каким-нибудь одним типом подключения. Разница между пассивным и активным FTP-подключением заключается в том, кто именно, клиент или сервер, инициирует

подключение к данным. В пассивном режиме клиент инициирует подключение к данным на сервере (по умолчанию через порт 20), а в активном режиме сервер инициирует подключение к данным (здесь порт по умолчанию не используется). Ни один из типов подключений не обладает какой-нибудь повышенной безопасностью. Более распространено пассивное подключение, поскольку клиент, защищенный NAT-брандмауэром, может подключиться к пассивному серверу, а также потому, что расширяемый пассивный сервер проще запрограммировать.

Автоматический вход в систему

Специфичную для FTP-сервера информацию об имени пользователя и пароле можно сохранить, чтобы ее не приходилось вводить при каждом посещении FTP-сайта. Каждая строка файла ~/.netrc идентифицирует сервер. При подключению к FTP-серверу ftp читает ~/.netrc для определения, нет ли у вас набора для автоматического входа на этот сервер. Формат строки ~/.netrc имеет следующий вид:

machine ***сервер*** *login* ***имя_пользователя*** *password* ***пароль***

где ***сервер*** — это имя сервера, ***имя_пользователя*** — ваше имя пользователя, а ***пароль*** — это ваш пароль на сервере. В последней строке файла следует поставить `default` вместо *machine*, чтобы указать имя пользователя и пароль для систем, не перечисленных в ~/.netrc. Строку `default` можно использовать для входа на анонимные серверы. Пример файла ~/.netrc имеет следующий вид:

```
$ cat ~/.netrc
machine bravo login max password mypassword
default login anonymous password max@example.com
```

Чтобы защитить информацию об учетных записях, имеющуюся в файле .netrc, нужно предоставить права на его чтение только тому пользователю, в чьем личном каталоге он находится. Дополнительные сведения о файле netrc даны на посвященной ему man-странице.

Примеры

Рассмотрим два сеанса использования ftp, в течение которых пользователь Max осуществляет передачу файлов на FTP-сервер по имени bravo и получение файлов с этого сервера. Когда Max вводит команду `ftp bravo`, локальный ftp-клиент подключается к серверу, который запрашивает имя пользователя и пароль. Поскольку он вошел в свою локальную систему под именем `max` ftp предлагает войти ему на сервер браво под именем `max`. Для входа под именем `max` Max может просто нажать **ВВОД**. Его имя на сервере bravo — `watson`, поэтому он набирает в ответ на приглашение на ввод имени `Name(bravo:max):` **watson**. В ответ на приглашение на ввод пароля `Password:` Max отвечает вводом обычного (удаленного) системного пароля, и FTP-сервер приветствует его и сообщает, что он использует двоичный режим передачи файлов (Using binary mode to transfer files). При работе ftp в двоичном режиме Max может передавать как ASCII-, так и двоичные файлы.

Подключение и вход в систему

```
$ ftp bravo
Connected to bravo.
220 (vsFTPd 2.0.7)
530 Please login with USER and PASS.
```

```
530 Please login with USER and PASS.
KERBEROS_V4 rejected as an authentication type
Name (bravo:max): watson
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

После входа на сервер Max использует ftp-команду ls для вывода содержимого своего удаленного рабочего каталога, который представлен его личным каталогом на сервере bravo. Затем он меняет рабочий каталог на memos и выводит имена находящихся в нем файлов.

ls и cd

```
ftp> ls
227 Entering Passive Mode (192,168,0,6,79,105)
150 Here comes the directory listing.
drwxr-xr-x    2 500      500          4096 Oct 10 23:52 expenses
drwxr-xr-x    2 500      500          4096 Oct 10 23:59 memos
drwxrwxr-x   22 500      500          4096 Oct 10 23:32 tech
226 Directory send OK.

ftp> cd memos
250 Directory successfully changed.

ftp> ls
227 Entering Passive Mode (192,168,0,6,114,210)
150 Here comes the directory listing.
-rw-r--r--    1 500      500          4770 Oct 10 23:58 memo.0514
-rw-r--r--    1 500      500          7134 Oct 10 23:58 memo.0628
-rw-r--r--    1 500      500          9453 Oct 10 23:58 memo.0905
-rw-r--r--    1 500      500          3466 Oct 10 23:59 memo.0921
-rw-r--r--    1 500      500          1945 Oct 10 23:59 memo.1102
226 Directory send OK.
```

Затем Max использует ftp-команду get для копирования файла memo.1102 с сервера на локальную систему. Использование им двоичного режима гарантирует получение полноценной копии файла независимо от его формата, будь он двоичным или же ASCII. Сервер подтверждает успешное копирование файла и отмечает размер файла и время, затраченное на его копирование. Затем Max копирует локальный файл memo.1114 на удаленную систему. Файл копируется в его удаленный рабочий каталог memos.

get и put

```
ftp> get memo.1102
local: memo.1102 remote: memo.1102
227 Entering Passive Mode (192,168,0,6,194,214)
150 Opening BINARY mode data connection for memo.1102 (1945 bytes).
226 File send OK.
1945 bytes received in 7.1e-05 secs (2.7e+04 Kbytes/sec)

ftp> put memo.1114
```

```
local: memo.1114 remote: memo.1114
227 Entering Passive Mode (192,168,0,6,174,97)
150 Ok to send data.
226 File receive OK.
1945 bytes sent in 2.8e-05 secs (6.8e+04 Kbytes/sec)
```

Вскоре Max принимает решение о том, что ему нужно скопировать все файлы, находящиеся в каталоге memos сервера bravo, в новый каталог на локальной системе. Он вводит команду ls, чтобы убедиться, что собирается копировать нужные ему файлы, но утилита ftp уже исчерпала лимит времени. Вместо выхода из ftp и ввода еще одной команды ftp из оболочки, Max дает ftp команду open bravo для повторного подключения к серверу. После входа в систему он использует ftp-команду cd для изменения рабочего каталога на сервере на каталог memos.

Исчерпание лимита времени и open

```
ftp> ls
421 Timeout.
Passive mode refused.
ftp> open bravo
Connected to bravo (192.168.0.6).
220 (vsFTPd 1.1.3)
...
ftp> cd memos
250 Directory successfully changed.
```

lcd (локальная cd)

К этому времени Max обнаруживает, что не создал новый каталог для хранения файлов, которые он хочет загрузить. Если дать ftp-команду mkdir, будет создан новый каталог на сервере, но пользователю Max нужен новый каталог на локальной системе. Он использует восклицательный знак (!), за которым следует команда mkdir memos.hold для вызова оболочки и запуска mkdir на локальной системе, создавая таким образом каталог по имени memos.hold в своем рабочем каталоге на локальной системе. (Имя своего рабочего каталога на локальной системе можно вывести с помощью команды !pwd.) Затем, поскольку ему нужно скопировать файлы с сервера в каталог memos.hold на локальной системе, Max должен сменить свой рабочий каталог на локальной системе. Если ввести команду !cd memos.hold, она не осуществит задуманное пользователем Max, потому что восклицательный знак приводит к вызову новой оболочки на локальной системе, и команда cd будет действовать только в этой новой оболочке, которая не является той оболочкой, под управлением которой работает утилита ftp. Для данной ситуации ftp предоставляет команду lcd (локальную cd), которая изменяет рабочий каталог для ftp и сообщает о новом локальном рабочем каталоге.

```
ftp> !mkdir memos.hold

ftp> lcd memos.hold
Local directory now /home/max/memos.hold
```

Для копирования всех файлов из удаленного каталога memos в каталог memos.hold на локальной системе Max использует ftp-команду mget (multiple get — множественное получение), за которой следует групповой символ звездочка (*). Когда ftp выводит приглашение для первого файла, Max понимает, что он забыл отключить вывод приглашений, поэтому в ответ на второе приглашение он вводит n и нажимает

комбинацию CONTROL+C для остановки копирования файлов. Сервер проверяет, не хочет ли он продолжить работу с вызванной им командой mget.

Затем Max выводит ftp-команду prompt, которая переключает вывод приглашения (выключает его, если это действие включено, и включает, если оно выключено). Теперь, когда он вводит команду `mget *`, ftp копирует все файлы, без вывода приглашения на подтверждение их копирования.

После получения всех требуемых файлов, Max вводит команду quit чтобы закрыть подключение к серверу, выйти из ftp и вернуться к приглашению локальной оболочки.

mget и prompt

```
ftp> mget *
mget memo.0514? n
mget memo.0628? CONTROL+C
Continue with mget? n
ftp> prompt
Interactive mode off.

ftp> mget *
local: memo.0514 remote: memo.0514
227 Entering Passive Mode (192,168,0,6,53,55)
150 Opening BINARY mode data connection for memo.0514 (4770 bytes).
226 File send OK.
4770 bytes received in 8.8e-05 secs (5.3e+04 Kbytes/sec)
local: memo.0628 remote: memo.0628
227 Entering Passive Mode (192,168,0,6,65,102)
150 Opening BINARY mode data connection for memo.0628 (7134 bytes).
226 File send OK.
...
150 Opening BINARY mode data connection for memo.1114 (1945 bytes).
226 File send OK.
1945 bytes received in 3.9e-05 secs (4.9e+04 Kbytes/sec)
ftp> quit
221 Goodbye.
```

gawk

Поиск и обработка шаблонов в файле.

gawk [ключи] [программа] [список_файлов]
gawk [ключи] –f файл_программы [список_файлов]

AWK является языком сканирования и обработки шаблонов, который осуществляет в одном или нескольких файлах поиск записей (обычно строк), соответствующих указанным шаблонам. Он обрабатывает строки путем выполнения таких действий, как вывод записей в стандартный вывод или увеличение показаний счетчика при каждом найденном соответствии. В противоположность процедурным языкам, AWK является языком, управляемым данными: вы описываете данные, с которыми нужно работать, и сообщаете AWK, что нужно делать с этими данными, как только они будут найдены.

СВЕДЕНИЯ О GAWK ДАНЫ В ГЛАВЕ 12

Сведения о реализациях языка AWK — awk, gawk и mawk даны в главе 12.

gcc

Компиляция программ на C и C++.

gcc [ключи] список_файлов [–язык]
g++ [ключи] список_файлов [–язык]

В операционных системах Linux и Mac OS X для обработки, компиляции, сборки и компоновки исходных файлов, написанных на языке C, используется GNU C-компилятор gcc. Такой же компилятор, но с другим внешним интерфейсом, g++, обрабатывает исходные программы на языке C++. Компиляторы gcc и g++ могут также осуществлять сборку и компоновку исходных программ на языке ассемблера, компоновку одних только объектных файлов или создание объектных файлов для их использования в общих библиотеках.

Эти компиляторы получают ввод из файлов, указанных в командной строке. Пока не будет использован ключ `-o`, они сохраняют исполняемую программу в файле a.out.

Компиляторы gcc и g++ являются частью GCC, коллекции компиляторов GNU Compiler Collection, которая включает программы пользовательского интерфейса для языков C, C++, Objective C, Fortran, Java и Ada, а также для библиотек, предназначенных для этих языков. Дополнительные сведения даны на сайте gcc.gnu.org.

GCC и G++

Хотя этот раздел посвящен компилятору gcc, многое в нем применимо также и к компилятору g++.

Аргументы

Список_файлов — это список файлов, обрабатываемых gcc.

Ключи

Gcc допускает работу без использования ключей с исходными файлами, написанными на языке C, ассемблер, объектными файлами и другими файлами, описание которых приведено в табл. V.16. Утилита gcc осуществляет соответствующую предварительную обработку, компиляцию, сборку и компоновку этих файлов, создавая исполняемый файл по имени a.out. Если gcc используется для создания объектного файла без его компоновки с целью создания исполняемого файла, каждый объектный файл получает имя путем добавления расширения .o к базовому имени соответствующего исходного файла. Если gcc используется для создания исполняемого файла, она удаляет объектные файлы после компоновки.

В данном разделе перечислены наиболее востребованные ключи. Когда с ключом связано определенное расширение имени файла, можно предположить, что gcc добавляет расширение к базовому имени исходного файла.

`-c` (compile — компилировать). Подавление компоновочной стадии компиляции. Компилирование и (или) сборка файлов исходного кода, и сохранение объектного кода в файлах с расширением .o.

`-D`*имя* `[=`*значение*`]` Обычно директива препропроцессора `#define` дается в файлах заголовков или включаемых файлах. Этот ключ можно использовать для

определения символических имен не в них, а в командной строке. Например, -DLinux является эквивалентом помещения во включаемый файл строки #define Linux, а -DMACH=i586 — то же самое, что и #define MACH i586.

-E (everything — все). Для исходных файлов программы подавление всех стадий компиляции, за исключением предварительной обработки (работы препроцессора) и записи результата в стандартный вывод. По соглашению, расширение .i используется для предварительно обработанного исходного кода на языке C, а расширение .ii используется для такого же кода на языке C++.

-fpic Принуждение gcc к созданию позиционно-независимого кода, подходящего для установки в библиотеку общего пользования.

-fwritable-strings По умолчанию GNU C-компилятор помещает строковые константы в защищенную память, где они не могут быть изменены. Некоторые (обычно устаревшие) программы предполагают возможность изменения строковых констант. Этот ключ изменяет поведение gcc, позволяя изменять строковые константы.

-g (gdb). Вставка диагностической информации в объектные файлы. Эта информация используется такими символьными отладчиками, как gdb. Хотя этот ключ рассчитан только на последующее использование отладчика, его безусловное включение в командную строку является правилом хорошего тона.

-I*каталог* Поиск включаемых файлов в *каталоге* перед их поиском в стандартных местах. Для поиска более чем в одном *каталоге* нужно воспользоваться этим ключом несколько раз.

-l*arg* (буква «l» в нижнем регистре). Поиск в каталогах /lib и /usr/lib библиотечного файла по имени lib*arg*.a. Если такой файл будет найден, gcc будет вести поиск любой востребованной функции в этой библиотеке. Аргумент *arg* следует заменить именем библиотеки, в которой нужно осуществлять поиск. Например, ключ –lm обычно связан со стандартной математической библиотекой libm.a. Позиция этого ключа играет весьма существенную роль: как правило, нужно, чтобы он появлялся в конце командной строки, но он может повторяться несколько раз для поиска в разных библиотеках. Библиотеки просматриваются в том порядке, в котором они фигурируют в командной строке. Компоновщик использует библиотеки только для того, чтобы разобраться с неопределенными символами из модулей, предшествующих библиотечному ключу в командной строке. Для просмотра lib*arg*.a можно использовать другие библиотечные пути, используя ключ -L.

-L*каталог* Добавление *каталога* к списку каталогов, в которых ведется поиск библиотек, задаваемых ключом -l. Каталоги, добавляемые в перечень с помощью ключа -L, просматриваются перед тем, как gcc ведет поиск в стандартных местах хранения библиотек.

-o *файл* (output — вывод). Задание имени исполняемой программе, получаемой в результате компоновки файла, вместо имени a.out.

-O*n* (optimize — оптимизация). Осуществление попытки улучшить (оптимизировать) объектный код, создаваемый компилятором. Аргумент *n* может прини-

мать значения 0, 1, 2 или 3 (или 06, если компилируется код для ядра Linux). По умолчанию для *n* принимается значение 1. Чем выше значение *n*, тем лучше оптимизация, но при этом может возрасти размер объектного файла и время работы gcc. Для выключения оптимизации нужно воспользоваться ключом -O0. Многие родственные ключи осуществляют тонкое управление типом оптимизации, которую пытается провести gcc при использовании ключа -O. Дополнительные сведения даны на info-странице, посвященной gcc.

-pedantic Воспринимаемый GNU C-компилятором язык C включает особенности, не являющиеся частью ANSI-стандарта для языка C. Использование этого ключа заставляет gcc отклонять эти расширения языка и воспринимать только свойства, присущие стандартному языку программирования C.

-Q Вывод имен функций по мере их компиляции, проводимой gcc. Этот ключ также выводит статистику, относящуюся к каждому проходу.

-S (suppress — подавление). Подавление стадий сборки и компоновки при компиляции файлов с исходным кодом. Получающиеся файлы на языке ассемблера имеют расширения имен файлов .s.

-traditional Принуждение gcc к восприятию только тех свойств языка программирования C, которые присутствуют в традиционном языке C, описанном Керниганом и Ричи (Kernighan и Ritchie). Использование этого ключа позволяет проводить правильную компиляцию устаревших программ, написанных с использованием традиционного языка C (существовавшего до появления ANSI-стандарта языка C).

-Wall Принуждение gcc к выводу предупреждений о вызывающем вопросы коде в файлах с исходным кодом. Многие родственные ключи позволяют осуществлять более тонкое управление предупредительными сообщениями.

Примечания

Предшествующий перечень ключей представляет собой лишь небольшую часть их полного набора, доступного при работе с GNU C-компилятором. Их полный список приведен на info-странице, посвященной gcc. Хотя ключ -o обычно используется для указания имени файла, в котором следует хранить объектный код, этот ключ можно также применять для указания имен файлов, создаваемых в результате других стадий компиляции. В следующем примере ключ -o заставляет ассемблерный код, создаваемый командой gcc, сохраняться в файле acode, а не в файле pgm.s, используемом по умолчанию:

```
$ gcc -S -o acode pgm.c
```

Утилита lint, имеющаяся у многих UNIX-систем, в Linux или Mac OS X недоступна. Тем не менее ключ -Wall осуществляет множество тех же самых проверок и может использоваться взамен lint.

В табл. V.16 дается сводка соглашений, используемых C-компилятором для присваивания расширений именам файлов.

Таблица V.16. Расширения имен файлов

Расширение	Тип файла
.a	Библиотека объектных модулей
.c	Исходный файл на языке C
.C, .cc или .cxx	Исходный файл на языке C++
.i	Исходный файл на языке C, обработанный препроцессором
.ii	Исходный файл на языке C++, обработанный препроцессором
.o	Объектный файл
.s	Исходный файл на языке ассемблера
.S	Исходный файл на языке ассемблера, требующий обработки препроцессором

Примеры

В первом примере осуществляется компиляция сборки и компоновки одиночной C-программы compute.c. Создаваемая исполняемая программа сохраняется в файле a.out. Утилита gcc удаляет объектный файл.

```
$ gcc compute.c
```

В следующем примере осуществляется компиляция той же программы с использованием оптимизатора C (ключа -O). Этот пример осуществляет сборку и компоновку оптимизированного кода. Ключ -o заставляет gcc сохранить исполняемую программу в файле compute.

```
$ gcc -O -o compute compute.c
```

В следующем примере компилируются, проходят сборку и компоновку исходный файл на языке C, файл на языке ассемблера и объектный файл. Исполняемая программа сохраняется в файле progo.

```
$ gcc -o progo procom.c profast.s proout.o
```

В следующем примере gcc просматривает стандартную математическую библиотеку, которая находится в каталоге /lib/libm.a при компоновке программы himath и сохраняет исполняемую программу в файле a.out:

```
$ gcc himath.c -lm
```

В следующем примере C-компилятор компилирует файл topo.c с ключами, заставляющими проводить проверку кода на наличие сомнительных фрагментов (ключ -Wall), не отвечающих к тому же ANSI-стандарту языка C (ключ -pedantic). Ключ -g приводит к появлению вставок, содействующих отладке исполняемого файла, который сохраняется в файле topo благодаря ключу -o topo. С помощью ключа -O3 разрешено проведение полной оптимизации кода.

Предупреждения, выдаваемые C-компилятором, отправляются на стандартный вывод. В данном примере первое и последнее предупреждения связаны с использованием ключа -pedantic, а другие предупреждения связаны с применением ключа -Wall.

```
$ gcc -Wall -g -O3 -pedantic -o topo topo.c
In file included from topo.c:2:
/usr/include/ctype.h:65: warning: comma at end of enumerator list
topo.c:13: warning: return-type defaults to 'int'
topo.c: In function 'main':
```

```
topo.c:14: warning: unused variable 'c'
topo.c: In function 'getline':
topo.c:44: warning: 'c' might be used uninitialized in this function
```

При компиляции программ, зависящих от включаемых файлов и библиотек X11, может понадобиться использование ключей -I и -L, чтобы сообщить gcc, где находятся эти включаемые файлы и библиотеки. В следующем примере эти ключи используются, чтобы сообщить gcc о необходимости компоновки программы с базовой библиотекой X11:

```
$ gcc -I/usr/X11R6/include plot.c -L/usr/X11R6/lib -lX11
```

GetFileInfo (OS X)

Вывод атрибутов файла.

GetFileInfo [ключ] файл

Утилита GetFileInfo выводит атрибуты файлов, включая тип файла и код создателя, время создания и последнего изменения и флаги атрибутов, такие как флаги невидимости и блокировки.

Аргументы

Файл указывает на одиночный файл или каталог, о котором GetFileInfo выводит информацию.

Ключи

Ключи для GetFileInfo соответствуют ключам для утилиты SetFile (см. соответствующий раздел). Без ключа GetFileInfo сообщает о метаданных ***файла***, показывая установленные флаги, тип файла и коды создателя, а также даты создания и последнего изменения. Отсутствующие данные опускаются. При использовании ключа GetFileInfo выводит информацию, определяемую только этим ключом. Эта утилита допускает применение только одного ключа, молча игнорируя лишние ключи.

-a*флаг* (attribute — атрибут). Отчет о состоянии отдельного флага атрибута по имени *флаг*. Этот ключ приводит к выводу 1, если флаг установлен, и 0, если флаг не установлен. ***Флаг*** должен следовать сразу же за -a, без промежуточных пробелов.

-c (creator — создатель). Вывод информации о коде создателя файла. Если файл является каталогом и не имеет кода создателя, этот ключ приводит к выводу сообщения об ошибке.

-d (date — дата). Вывод даты создания файла в формате 24-часовой шкалы мм/дд/гггг чч:мм:сс.

-m (modification — изменение). Вывод даты последнего изменения файла в формате мм/дд/гггг чч:мм:сс с использованием 24-часовой шкалы.

-P (без разыменования). Для каждого файла, являющегося символьной ссылкой, вывод информации о символьной ссылке, а не о файле, на который она

указывает. Действие этого ключа распространяется на все файлы. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример использования этого ключа приведен в разделе «chgrp», в подразделе «Примеры».

`-t` (type — тип). Вывод типа кода файла. Если файл является каталогом и не имеет типа кода, этот ключ приводит к выводу сообщения об ошибке.

Комментарии

Без ключа утилита GetFileInfo выводит флаги в виде строки `avbstclinmedz`, в которой буквы в верхнем регистре указывают на то, что флаг установлен.

Примечания

Для установки атрибутов файла можно воспользоваться утилитой SetFile (см. соответствующий раздел). Права доступа и сведения о владельце файлов в Mac OS X (см. раздел «Права доступа» на с. 114) можно установить с помощью утилиты chmod или утилиты chown, описания которых даны в соответствующих разделах, а вывести эту информацию можно с помощью утилиты ls или утилиты stat, описания которых также даны в соответствующих разделах. Каталоги не имеют кодов типа или создателя и у них могут быть не все флаги. Утилита GetFileInfo не может читать информацию о специальных файлах, например о файлах устройств.

Примеры

В первом примере показан вывод утилиты GetFileInfo при ее вызове без ключа.

```
$ GetFileInfo picture.jpg
file: "/private/tmp/picture.jpg"
type: "JPEG"
creator: "GKON"
attributes: avbstClinmedz
created: 07/18/2005 15:15:26
modified: 07/18/2005 15:15:26
```

Единственной буквой в верхнем регистре в строке атрибутов является буква `C`, показывающая, что данный флаг установлен. Флаг `c` сообщает утилите Finder о необходимости поиска для этого файла пользовательского значка.

В следующем примере используется ключ `-a`, приводящий к выводу флагов атрибутов файла:

```
$ GetFileInfo -a /Applications/Games/Alchemy/Alchemy
avBstclInmedz
```

Выведенная информация показывает, что установлены флаги `b` и `i`.

Утилита GetFileInfo может обрабатывать при каждом вызове только один файл. В следующей составной bash-команде для вывода кода создателя для нескольких файлов используется цикл for (см. раздел «Управляющая структура for...in» на с. 429). Команда echo выводит имя просматриваемого файла, поскольку GetFileInfo никогда не выводит имя файла:

```
$ for i in *
> do echo -n "$i: "; GetFileInfo -c "$i"
> done
Desktop: Desktop is a directory and has no creator
   Desktop: Desktop является каталогом и не имеет создателя
Documents: Documents is a directory and has no creator
   Documents: Documents является каталогом и не имеет создателя
…
aa: ""
ab: ""
…
```

grep

Поиск шаблонов в файлах.

grep [ключи] шаблон [список_файлов]

Утилита grep ведет построчный поиск в одном или нескольких текстовых файлах, пользуясь шаблоном, который может быть простой строкой или другой какой-нибудь формой регулярного выражения. Утилита grep осуществляет различные действия, определяемые ключами, всякий раз, когда находит строку, содержащую соответствие шаблону. Эта утилита получает ввод либо из файлов, указанных в командной строке, либо со стандартного ввода.

Аргументы

Шаблон — это регулярное выражение, которое нужно процитировать, если оно содержит специальные символы, пробелы или символы табуляции. Проще всего процитировать эти символы, взяв все выражение в одинарные кавычки.

Список_файлов — это список путевых имен обычных текстовых файлов, просматриваемых утилитой grep. При использовании ключа `-r` ***список_файлов*** может включать в себя каталоги с просматриваемым содержимым.

Ключи

При использовании любых ключей утилита grep отправляет строки, содержащие соответствие шаблону, на стандартный вывод. Если в командной строке указывается более одного файла, grep ставит перед каждой выводимой строкой имя файла, из которого она поступила, за которым следует двоеточие.

Главные ключи

Одновременно можно использовать только один из трех следующих ключей. Обычно потребности в использовании этих ключей не возникает, потому что grep по умолчанию использует ключ `-G`, который вызывает обычный режим работы grep.

`-E` (extended — расширенное). Интерпретация шаблона в качестве расширенного регулярного выражения. Команда `grep -E` является аналогом использования утилиты **egrep**. См. раздел «Примечания».

`-F` (fixed — фиксированное). Интерпретация шаблона в качестве фиксированной строки символов. Команда `grep -F` является аналогом использования утилиты **fgrep**.

`-G` (grep). Интерпретация шаблона в качестве базового регулярного выражения. Если не указан другой главный ключ, то этот главный ключ используется по умолчанию.

Остальные ключи

Утилита **grep** допускает применение общих ключей, описание которых дано в табл. V.2.

ВЕРСИЯ GREP ДЛЯ MAC OS X ДОПУСКАЕТ ИСПОЛЬЗОВАНИЕ ДЛИННЫХ КЛЮЧЕЙ

Ключи для grep, перед которыми ставится двойной дефис (- -), работают как под управлением Linux, так и под управлением OS X.

`--count -c` Вывод только количества строк, содержащих соответствие шаблону в каждом файле.

`--context=`*n*
`-C` *n* Вывод *n* строк содержимого, окружающего строки, имеющие соответствие шаблону.

`--file=` ***файл***
`-f` ***файл*** Чтение файла, содержащего по одному шаблону в каждой строке, и поиск строк ввода, соответствующих каждому шаблону.

`--no-filename`
`-h` Подавление вывода имени файла в начале каждой строки, если просматривается несколько файлов.

`--ignore-case`
`-i` Принуждение к соответствию букв нижнего регистра в шаблоне буквам верхнего регистра в файл, и наоборот. Этот ключ используется при поиске слова, которое может появляться в начале предложения (то есть слово может начинаться с большой, а может начинаться и с малой буквы).

`--files-with-matches`
`-l` (буква «l» в нижнем регистре; list — список). Вывод только имени каждого файла, в котором имеется одно или несколько соответствий. Имя файла появляется только один раз, даже если в файле имеется более одного соответствия.

`--max-count=`*n*
`-m` *n* Прекращение чтения каждого файла или стандартного ввода после вывода *n* строк, содержащих соответствия.

`--line-number`
`-n` Установка перед каждой строкой ее номера в файле. Файл при этом не обязательно должен иметь нумерацию строк.

`--quiet` ***или***
`--silent`
`-q` Отмена любого вывода в стандартный вывод, установка только кода выхода.

`--recursive`
`-r` ***или*** `-R` Рекурсивный спуск по каталогам, указанным в ***списке_файлов***, и обработка файлов, находящихся в этих каталогах.

`--no-messages` `-s` (silent — молчание). Отмена вывода сообщений об ошибках, если файл из ***списка_файлов*** не существует или для него отсутствует право на чтение.

`--invert-match` `-v` Принуждение к соответствию условиям поиска тех строк, в которых отсутствуют соответствия шаблону. При использовании только этого ключа grep выводит все строки, в которых не содержится соответствия шаблону.

`--word-regexp` `-w` При использовании этого ключа шаблон должен соответствовать всему слову целиком. Этот ключ применяется при ведении поиска конкретного слова, которое может также появиться в файле в качестве подстроки какого-нибудь другого слова.

`--line-regexp` `-x` Принуждение к соответствию шаблона только всей строке целиком.

Примечания

Утилита grep возвращает статус выхода 0, если соответствие найдено, 1, если соответствие не найдено, и 2, если файл недоступен или в команде grep содержится синтаксическая ошибка.

egrep и fgrep Такие функции, как и у grep, выполняются еще двумя утилитами. Утилита egrep (аналог `grep -E`) позволяет использовать расширенные регулярные выражения, включающие различные наборы специальных символов, не использующиеся в базовом регулярном выражении. Утилита fgrep (аналог `grep -F`) меньше по размеру и более быстрая в работе, но обрабатывает только простые строки, а не регулярные выражения.

GNU grep, работающая под управлением Linux и Mac OS X, использует вместо базовых расширенные регулярные выражения. Поэтому egrep фактически то же самое, что и grep. Незначительные различия рассматриваются на info-странице, посвященной grep.

Примеры

В следующих примерах предполагается, что в рабочем каталоге содержатся три файла: testa, testb и testc.

```
Файл testa   Файл testb   Файл testc
aaabb        aaaaa        AAAAA
bbbcc        bbbbb        BBBBB
ff-ff        ccccc        CCCCC
cccdd        ddddd        DDDDD
dddaa
```

Утилита grep может искать соответствие шаблону, представляющему собой простую строку символов. В следующей командной строке ведется поиск в файле testa, и выводится каждая строка, содержащая строковое значение bb:

```
$ grep bb testa
aaabb
bbbcc
```

Ключ -v инвертирует смысл теста. В следующем примере выводятся строки из файла testa, которые не содержат bb:

```
$ grep -v bb testa
ff-ff
cccdd
dddaa
```

Ключ -n приводит к выводу номера каждой отображаемой строки:

```
$ grep -n bb testa
1:aaabb
2:bbbcc
```

Утилита grep может вести поиск более чем в одном файле. В данном примере ведется поиск в каждом файле, имеющемся в рабочем каталоге. Имя файла, содержащего строку, ставится перед каждой вводимой строкой.

```
$ grep bb *
testa:aaabb
testa:bbbcc
testb:bbbbb
```

Когда поиск строки bb ведется с ключом -w, grep ничего не выводит, поскольку ни в одном из файлов не содержится строка bb в виде отдельного слова:

```
$ grep -w bb *
$
```

Поиск, проводимый утилитой grep, ведется с учетом регистра букв. Поскольку в предыдущих примерах указан шаблон bb, состоящий из букв в нижнем регистре, grep не находит в файле testc строку BBBBB, состоящую из букв верхнего регистра. Ключ -i заставляет все буквы, и нижнего и верхнего регистра, соответствовать в шаблоне буквам обоих регистров:

```
$ grep -i bb *
testa:aaabb
testa:bbbcc
testb:bbbbb
testc:BBBBB
$ grep -i BB *
testa:aaabb
testa:bbbcc
testb:bbbbb
testc:BBBBB
```

Ключ -c приводит к выводу номеров строк в каждом файле, содержащем соответствие:

```
$ grep -c bb *
testa:2
testb:1
testc:0
```

Ключ -f ищет соответствия каждому из шаблонов, имеющемуся в файле шаблонов. В следующем примере в файле gfile содержатся два шаблона, по одному в каждой строке, и grep ищет соответствие шаблонам, имеющимся в gfile:

```
$ cat gfile
aaa
```

```
bbb

$ grep -f gfile test*
testa:aaabb
testa:bbbcc
testb:aaaaa
testb:bbbbb
```

В следующей командной строке проверяется файл text2 и выводятся строки, содержащие строку символов, начинающуюся с символов st, за которыми следует от нуля и более символов (.* представляет в регулярном выражении нуль и более символов), и заканчивающуюся на ing:

```
$ grep 'st.*ing' text2
...
```

Регулярное выражение ^, которое соответствует началу строки, может быть использовано и само по себе для соответствия каждой строке файла. Вместе с ключом -n, регулярное выражение ^ может использоваться для вывода строк в файле, перед которыми будут поставлены их номера:

```
$ grep -n '^' testa
1:aaabb
2:bbbcc
3:ff-ff
4:cccdd
5:dddaa
```

Следующая командная строка подсчитывает количество появлений инструкции #include в исходных файлах с программами на языке C, которые находятся в рабочем каталоге. Ключ -h заставляет grep подавлять вывод имен файлов. Ввод для команды sort состоит из всех строк из файлов *.c, которые соответствуют шаблону #include. Вывод из команды sort представляет собой отсортированный список строк, содержащих множество дубликатов. Когда утилита uniq с ключом -c обрабатывает этот отсортированный список, она выводит повторяющиеся строки только один раз, вместе со счетчиком количества повторений в ее вводе.

```
$ grep -h '#include' *.c | sort | uniq -c
9 #include "buff.h"
2 #include "poly.h"
1 #include "screen.h"
6 #include "window.h"
2 #include "x2.h"
2 #include "x3.h"
2 #include <math.h>
3 #include <stdio.h>
```

Последняя команда вызывает редактор vim со списком файлов, имеющихся в рабочем каталоге, в которых содержится строка Sampson. Команда $(...)является подстановочной конструкцией (см. раздел «Командная подстановка» на с. 358), заставляющей оболочку выполнять grep в этом месте и снабжать vim списком имен файлов, которые нужно отредактировать:

```
$ vim $(grep -l 'Sampson' *)
...
```

Одинарные кавычки в данном примере не нужны, но они понадобятся в том случае, если регулярное выражение, на основе которого ведется поиск, будет содержать специальные символы или пробелы. Лучше выработать привычку заключать шаблон в кавычки, чтобы оболочка не интерпретировала любой специальный символ, который может содержаться в шаблоне.

gzip

Сжатие и восстановление файлов.

gzip [ключи] [список_файлов]
gunzip [ключи] [список_файлов]
zcat [список_файлов]

Утилита gzip сжимает файлы, утилита gunzip восстанавливает файлы, сжатые утилитой gzip, а утилита zcat выводит файлы, сжатые утилитой gzip.

Аргументы

Список_файлов — это список, состоящий из имен одного или нескольких файлов, которые должны быть сжаты или восстановлены. Когда в ***списке_файлов*** появляется каталог и не используется ключ `--recursive`, утилиты gzip и gunzip выводят сообщение об ошибке и игнорируют каталог. При использовании ключа `--recursive` утилиты gzip и gunzip сжимают и восстанавливают файлы в рекурсивном режиме внутри иерархии каталогов. Если ***список_файлов*** пуст или если присутствует специальный ключ – (дефис), gzip читает данные со стандартного ввода. Ключ `--stdout` заставляет gzip и gunzip вести запись на стандартный вывод.

Информация этого раздела применима также к утилите gunzip, являющейся ссылкой на утилиту gzip.

Ключи

Утилиты gzip, gunzip и zcat допускают применение общих ключей, описание которых дано в табл. V.2.

ВЕРСИИ GZIP, GUNZIP И ZCAT ДЛЯ MAC OS X ДОПУСКАЮТ ИСПОЛЬЗОВАНИЕ ДЛИННЫХ КЛЮЧЕЙ

Ключи для gzip, gunzip и zcat, перед которыми ставится двойной дефис (– –), работают как под управлением Linux, так и под управлением OS X.

`--stdout -c` Запись результатов сжатия или восстановления на стандартный вывод, не в файл с именем *имя_файла.gz*.

`--decompress` *или* `--uncompress -d` Восстановление файла, сжатого утилитой gzip. Этот ключ, используемый с gzip, является эквивалентом команды gunzip.

`--force -f` Запись поверх существующего файла вывода сжатого (восстановленного) файла.

`--list -l` Для каждого сжатого файла в ***списке_файлов*** вывод размеров в сжатом и несжатом состоянии, степени сжатия и имени файла перед сжатием. Для отображения дополнительной информации этот ключ нужно использовать вместе с ключом `--verbose`.

`--fast` *или* `-n` `--best` Управление соотношением между скоростью и степенью сжатия. Аргумент *n* является числом в диапазоне от 1 до 9; уровень 1 соответствует наиболее быстрому и наименее плотному сжатию, а уровень 9 — наиболее плотному и наименее быстрому сжатию. По умолчанию используется уровень 6. Ключи `--fast` и `--best` являются синонимами ключей `-1` и `-9` соответственно.

`--quiet -q` Подавление предупреждающих сообщений.

`--recursive -r` Рекурсивный спуск по каталогам, указанным в ***списке_файлов***, и сжатие (восстановление) файлов в этих каталогах.

`--test -t` Проверка целостности сжатого файла. Если с файлом все в порядке, этот ключ ничего не выводит.

`--verbose -v` Вывод имени файла, имени сжатого файла и степени сжатия при обработке каждого файла.

Комментарии

Сжатие файлов сокращает дисковое пространство, необходимое для их хранения, и сокращает время, необходимое для их передачи между системами. Когда утилита gzip сжимает файл, она добавляет к его имени расширение .gz. Например, сжатие файла по имени fname приведет к созданию файла по имени fname.gz и, пока не будет использован ключ `--stdout` (`-c`), к удалению исходного файла. Для восстановления файла fname используется команда gunzip с аргументом `fname.gz`.

При сжатии с помощью утилиты gzip почти все файлы становятся намного меньше по размеру. В редких случаях файлы становятся чуть-чуть длиннее. Насколько файл станет меньше, определяется его типом и содержимым (а также ключом *-n*); текстовые файлы зачастую сокращаются на 60–70 процентов.

Такие атрибуты файла, как его владелец, права доступа к нему, а также время последнего обращения и изменения, остаются нетронутыми при его сжатии утилитой gzip и восстановлении с помощью утилиты gunzip. Если сжатая версия файла уже существует, gzip сообщает об этом факте и запрашивает подтверждение перед перезаписью существующего файла. Если файл содержит несколько ссылок на себя, gzip выдает сообщение об ошибке и прекращает работу. Ключ `--force` отменяет поведение по умолчанию в обеих рассмотренных ситуациях.

Примечания

Без ключа `--stdout` (`-c`) gzip удаляет файлы, перечисленные в ***списке_файлов***.

Утилита bzip2 (см. соответствующее описание) сжимает файлы более эффективно, чем утилита gzip.

Кроме формата gzip, утилита gunzip распознает ряд других форматов сжатия, что дает ей возможность восстанавливать файлы, сжатые утилитой compress.

Чтобы посмотреть на пример файла, который становится больше по размеру при сжатии утилитой gzip, можно сравнить размер файла, который уже был сжат, с тем же файлом, сжатым с помощью gzip еще раз. Поскольку gzip выражает недовольство, когда ей предоставляется аргумент с расширением .gz, перед повторным сжатием файл нужно переименовать.

Утилита tar с модификатором -z (см. раздел, посвященный tar), вызывает утилиту gzip.

Следующие родственные утилиты выводят содержимое сжатых файлов и производят с ними различные действия. Но при работе с файлами ни одна из них не изменяет сами файлы.

`zcat` *список_файлов* — Работает так же, как и утилита cat, за исключением того, что использует утилиту gunzip для восстановления файлов из *списка_файлов* по мере их копирования на стандартный вывод.

`zdiff` [*ключи*] *файл1* [*файл2*] — Работает так же, как и утилита diff (см. описание этой утилиты), за исключением того, что *файл1* и *файл2* восстанавливаются, если это необходимо, с помощью утилиты gunzip. Утилита zdiff допускает использование тех же ключей, что и утилита diff. Если опустить аргумент *файл2*, утилита zdiff сравнивает *файл1* со сжатой версией этого же файла.

`zless` *список_файлов* — Работает так же, как и утилита less, за исключением того, что она использует gunzip для восстановления файлов из *списка_файлов* по мере их вывода.

Примеры

В первом примере gzip сжимает два файла. Затем gunzip восстанавливает один из файлов. Когда файл сжимается и восстанавливается, его размер изменяется, но время последнего изменения остается прежним:

```
$ ls -l
total 175
-rw-rw-r-- 1 max group 33557 Jul 20 17:32 patch-2.0.7
-rw-rw-r-- 1 max group 143258 Jul 20 17:32 patch-2.0.8
$ gzip *
$ ls -l
total 51
-rw-rw-r-- 1 max group 9693 Jul 20 17:32 patch-2.0.7.gz
-rw-rw-r-- 1 max group 40426 Jul 20 17:32 patch-2.0.8.gz
$ gunzip patch-2.0.7.gz
$ ls -l
total 75
-rw-rw-r-- 1 max group 33557 Jul 20 17:32 patch-2.0.7
-rw-rw-r-- 1 max group 40426 Jul 20 17:32 patch-2.0.8.gz
```

В следующем примере файлы, находящиеся в личном каталоге пользователя по имени Sam, архивируются с помощью утилиты cpio (см. соответствующий раздел). Перед записью его на устройство, примонтированное к каталогу /dev/sde1, архив сжимается с помощью утилиты gzip.

```
$ find ~sam -depth -print | cpio -oBm | gzip >/dev/sde1
```

head

Вывод начала файла.

head [ключи] [список_файлов]

Утилита head выводит начало (заголовок) файла. Эта утилита получает ввод либо из одного или нескольких файлов, либо со стандартного ввода.

Аргументы

Список_файлов — это список, состоящий из путевых имен файлов, выводимых утилитой head. Если указывается более одного файла, head выводит перед несколькими строками каждого файла его имя. Если файл не указан, head получает ввод со стандартного ввода.

Ключи

При работе под управлением Linux утилита head допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. За исключением особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--bytes=`*n*`[`*u*`]`
`-c` *n*`[`*u*`]` Вывод первых *n* байтов (символов) файла. При работе только под управлением Linux аргумент *u* является необязательным мультипликативным суффиксом из числа тех, которые рассмотрены в разделе «Стандартные мультипликативные суффиксы», за исключением того, что утилита head использует для килобайтов (1024-байтных блоков) букву k в нижнем регистре и допускает применение буквы b для 512-байтных блоков. Если включается мультипликативный суффикс, утилита head ведет счет, используя вместо байтов соответствующие единицы измерения.

`--lines=`*n*
`-n` *n* Вывод первых *n* строк файла. Ключ *-n* можно использовать для указания *n* строк без использования ключевого слова `lines` или ключа `-n`. Если для *n* указывается отрицательное значение, утилита head выводит все, кроме последних *n* строк файла.

`--quiet -q` Подавление заголовочной информации при указании в командной строке более одного имени файла. (Linux)

Примечания

По умолчанию утилита head выводит первые десять строк файла.

Примеры

Примеры в этом разделе основаны на следующем файле:

```
$ cat eleven
строка один
строка два
```

```
строка три
строка четыре
строка пять
строка шесть
строка семь
строка восемь
строка девять
строка десять
строка одиннадцать
```

Без аргументов утилита head выводит первые десять строк файла:

```
$ head eleven
строка один
строка два
строка три
строка четыре
строка пять
строка шесть
строка семь
строка восемь
строка девять
строка десять
```

В следующем примере выводятся первые три строки (`-n 3`) файла:

```
$ head -n 3 eleven
строка один
строка два
строка три
```

Следующий пример является эквивалентом предыдущего:

```
$ head -3 eleven
строка один
строка два
строка три
```

В следующем примере выводятся первые восемь символов (`-c 8`) файла:

```
$ head -c 8 eleven
строка о$
```

В последнем примере выводятся все строки файла, кроме последних семи:

```
$ head -n -7 eleven
строка один
строка два
строка три
строка четыре
```

kill

Завершает процесс на основе его PID.

kill [ключ] PID-список

kill –l [имя_сигнала | номер_сигнала]

Утилита kill отправляет сигнал одному или нескольким процессам. Обычно этот сигнал завершает процессы.

Чтобы утилита kill работала, процессы должны принадлежать пользователю, запустившему kill на выполнение, с одним лишь исключением: пользователь, работающий с root-правами, может завершить любой процесс. Ключ `-l` (буква «l» в нижнем регистре) приводит к выводу информации о сигналах.

Аргументы

PID-список — это список идентификационных номеров процессов (PID), завершаемых с помощью утилиты kill.

Ключи

`-l` (list — список). В отсутствие аргументов вывод списка сигналов. С аргументом ***имя_сигнала***, вывод соответствующего ***номера_сигнала***. С аргументом ***номер_сигнала***, вывод соответствующего ***имени_сигнала***.

-имя_сигнала | -номер_сигнала Отправка сигнала, указанного с помощью ***имени_сигнала*** или ***номера_сигнала PID-списку***. ***Имя_сигнала*** можно указать с приставкой `SIG` или без нее (например, `SIGKILL` или `KILL`). Без этого ключа kill отправляет сигнал на программное завершение (`SIGTERM`; номер сигнала 15).

Примечания

См. также описание утилиты killall.

См. список сигналов в табл. 10.5. Команда `kill -l` выводит полный список номеров и имен сигналов.

В дополнение к утилите kill, в оболочках Bourne Again и TC Shells доступны встроенные команды kill. Эти встроенные команды работают точно так же, как рассматриваемая в данном разделе утилита. Для использования утилиты нужно дать команду /bin/kill, а для использования встроенной команды нужно ввести команду kill. Обычно то, какая именно версия вызвана, не имеет значения.

При запуске фонового процесса оболочка выводит его PID-номер. Чтобы определить PID-номера, можно также воспользоваться утилитой ps.

Если сигнал программного завершения не останавливает процесс, нужно попробовать отправить сигнал `KILL` (сигнал номер 9). Процесс может выбрать игнорирование любого сигнала, кроме `KILL`.

Утилита (встроенная команда) kill допускает вместо ***PID-списка*** использование идентификатора задания. Этот идентификатор состоит из знака процента (`%`), за которым следует либо номер задания, либо строка, способная идентифицировать задание уникальным образом.

Для завершения всех процессов, инициированных текущим процессом входа в систему, и принуждения операционной системы к вашему выходу из нее, нужно ввести команду `kill -9 0`.

РАБОТАЯ С ROOT-ПРАВАМИ, НЕ ЗАПУСКАЙТЕ KILL С АРУГМЕНТАМИ –9 0 ИЛИ KILL 0

Если при работе с root-правами запустить команду kill –9 0, система зависнет.

Примеры

В первом примере показана командная строка, выполняющая файл compute в качестве фонового процесса, и утилита kill, завершающая этот процесс:

```
$ compute &
[2] 259
$ kill 259
$ ВВОД
[2]+  Terminated              compute
```

В следующем примере показана утилита ps, с помощью которой определяется PID-номер фонового процесса, выполняющего программу по имени xprog, и утилита kill, завершающая xprog с помощью сигнала TERM:

```
$ ps
PID TTY          TIME CMD
 7525 pts/1    00:00:00 bash
14668 pts/1    00:00:00 xprog
14699 pts/1    00:00:00 ps

$ kill -TERM 14668
$
```

killall

Завершение процесса с использованием его имени.

killall [ключ] список_имен

Утилита killall отправляет сигнал одному или нескольким процессам, выполняющим определенные команды. Обычно этот сигнал приводит к завершению процессов. Чтобы утилита killall работала, процесс должен принадлежать пользователю, запустившему killall на выполнение, с одним лишь исключением: пользователь, работающий с root-правами, может завершить любой процесс.

Аргументы

Список_имен — это список имен программ, которые должны получить сигнал. В качестве разделителей в этом списке используются пробелы.

Ключи

Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--interactive` `-i` Вывод приглашения на подтверждение, перед тем как завершить процесс. (Linux)

`--list -l` Вывод списка сигналов (но `kill -l` выводит более приглядный список). С этим ключом killall не воспринимает ***список_имен***.

`--quiet -q` Отказ от вывода сообщения, если killall не в состоянии завершить процесс. (Linux)

`-имя_сигнала | -номер_сигнала` Отправка сигнала, указанного с помощью ***имени_сигнала*** или ***номера_сигнала списку_имен***. ***Имя_сигнала*** можно указать с приставкой `SIG` или без нее (например, `SIGKILL` или `KILL`). Без этого ключа kill отправляет сигнал на программное завершение (`SIGTERM`; номер сигнала 15).

Примечания

См. также описание утилиты kill.

См. список сигналов в табл. 10.5. Команда `kill -l` выводит полный список номеров и имен сигналов.

Если сигнал программного завершения не останавливает процесс, нужно попробовать отправить сигнал `KILL` (сигнал номер 9). Процесс может выбрать игнорирование любого сигнала, кроме `KILL`.

Чтобы определить имя программы, выполнение которой нужно завершить, следует воспользоваться утилитой ps (см. соответствующий раздел).

Примеры

Чтобы провести эксперимент с утилитой killall, можно ввести следующие команды:

```
$ sleep 60 &
[1] 23274
$ sleep 50 &
[2] 23275
$ sleep 40 &
[3] 23276
$ sleep 120 &
[4] 23277
$ killall sleep
$ ВВОД
[1]   Terminated              sleep 60
[2]   Terminated              sleep 50
[3]-  Terminated              sleep 40
[4]+  Terminated              sleep 120
```

Следующая команда, запущенная пользователем, имеющим root-права, завершает работу всех экземпляров браузера Firefox:

```
# killall firefox
```

launchctl (OS X)

Управление демоном (программой, работающей в фоновом режиме) launchd.

*launchctl **[команда[ключи] [аргументы]]***

Утилита launchctl управляет работой демона launchd.

Аргументы

Команда — это команда, которую launchctl отправляет демону launchd. В табл. V.17 перечислены некоторые команды и ключи, а также аргументы, допускаемые каждой командой. Без ***команды*** утилита launchctl построчно читает команды, ключи и аргументы со стандартного ввода. Без ***команды***, когда стандартный ввод поступает с клавиатуры, утилита launchctl работает в интерактивном режиме.

Таблица V.17. Команды launchctl

Команда	Аргумент	Описание
`help`	Отсутствует	Вывод справки
`list`	Отсутствует	Вывод списка заданий, загруженных в launchd
`load [-w]`	Конфигурационный файл задания	Загрузка задания, имя которого определено с помощью аргумента
`shutdown`	Отсутствует	Подготовка к остановке путем удаления всех заданий
`start`	Имя задания	Запуск задания, имя которого определено с помощью аргумента
`stop`	Имя задания	Остановка задания, имя которого определено с помощью аргумента
`unload [-w]`	Конфигурационный файл задания	Выгрузка задания, имя которого определено с помощью аргумента

Ключ

Ключ используется только командами загрузки — `load` и выгрузки — `unload`.

`-w` (write — запись). При загрузке файла удаление ключа `Disabled`, и сохранение измененного конфигурационного файла. При выгрузке файла добавление ключа `Disabled` и сохранение измененного конфигурационного файла.

Комментарии

Утилита launchctl является пользовательским интерфейсом для работы с launchd, который управляет системными демонами и фоновыми задачами (называемыми заданиями). Каждое задание описывается конфигурационным файлом задания, который является файлом списка свойств, имеющим формат, определяемый на man-странице, посвященной `launchd.plist`.

Из соображений безопасности пользователи, не работающие с root-правами, не могут связываться с первичным системным процессом launchd, PID 1. Когда один из таких пользователей загружает задания, OS X создает для этого пользователя новый экземпляр launchd. После выгрузки всех его заданий этот экземпляр демона launchd прекращает свою работу.

Примечания

Утилита launchctl и демон launchd были представлены в Mac OS X версии 10.4. При работе под управлением версии 10.3 и более ранних версий системные задания управлялись утилитами init, xinetd и cron.

Примеры

В первом примере, который запускается пользователем, работающим с root-правами, используется команда `list` для вывода списка заданий, запущенных на локальной системе:

```
# launchctl list
PID     Status  Label
51479   -       0x109490.launchctl
50515   -       0x10a780.bash
50514   -       0x10a680.sshd
50511   -       0x108d20.sshd
22      -       0x108bc0.securityd
-       0       com.apple.launchctl.StandardIO
37057   -       [0x0-0x4e84e8].com.apple.ScreenSaver.Engine
27860   -       0x10a4e0.DiskManagementTo
27859   -       [0x0-0x3a23a2].com.apple.SoftwareUpdate
…
```

В следующем примере разрешается работа сервиса ntalk. Просмотр файла ntalk.plist до и после выполнения команды launchctl показывает, что launchctl изменила файл, удалив ключ `Disabled`.

```
# cat /System/Library/LaunchDaemons/ntalk.plist
…
<dict>
        <key>Disabled</key>
        <true/>
        <key>Label</key>
        <string>com.apple.ntalkd</string>
…
# launchctl load -w /System/Library/LaunchDaemons/ntalk.plist
# cat /System/Library/LaunchDaemons/ntalk.plist
…
<dict>
        <key>Label</key>
        <string>com.apple.ntalkd</string>
…
```

Без аргументов утилита launchctl выводит приглашение на ввод команды со стандартного ввода. Чтобы выйти из launchctl, следует ввести команду `quit` или нажать комбинацию CONTROL+D. В последнем примере пользователь, работающий с root-правами, заставляет launchctl вывести список заданий, а затем остановить задание, которое запускает airportd:

```
# launchctl
launchd% list
PID     Status  Label
8659    -       0x10ba10.cron
1       -       0x10c760.launchd
…
-       0       com.apple.airport.updateprefs
-       0       com.apple.airportd
-       0       com.apple.AirPort.wps
-       0       0x100670.dashboardadvisoryd
```

```
-       0       com.apple.launchctl.System
launchd% stop com.apple.airportd
launchd% quit
```

less

Поэкранный вывод текстовых файлов.

less [ключи] [список_файлов]

Утилита less осуществляет поэкранный вывод содержимого текстовых файлов.

Аргументы

Список_файлов — это список тех файлов, которые нужно просмотреть. Если ***список_файлов*** не используется, less читает данные со стандартного ввода.

Ключи

Утилита less допускает применение общих ключей, описание которых дано в табл. V.2.

ВЕРСИЯ LESS ДЛЯ MAC OS X ДОПУСКАЕТ ПРИМЕНЕНИЕ ДЛИННЫХ КЛЮЧЕЙ

Ключи для less, перед которыми ставится двойной дефис (– –), работают как под Mac OS X, так и под Linux.

`--clear-screen` `-c` Вместо прокрутки перерисовка экрана от верхней строки вниз.

`--QUIT-AT-EOF` `-E` (exit — выход). Обычно less для завершения работы требует введения команды q. Этот ключ приводит к автоматическому выходу из программы при первом же чтении конца файла.

`--quit-at-eof` `-e` (exit — выход). Действие ключа похоже на действие -E, за исключением того, что less осуществляет автоматический выход при втором чтении конца файла.

`--quit-if-one-screen` `-F` Вывод файла и выход, если файл может быть отображен на одном экране.

`--ignore-case` `-i` Принуждение к поиску, при котором строка с буквами в нижнем регистре будет соответствовать буквам как в верхнем, так и в нижнем регистре. Этот ключ игнорируется, если указывается шаблон, включающий буквы верхнего регистра.

`--IGNORE-CASE` `-I` Принуждение к поиску, при котором строка с буквами в любом регистре будет соответствовать буквам как в верхнем, так и в нижнем регистре, независимо от регистра шаблона поиска.

`--long-prompt` `-m` В каждое приглашение включается отчет о проценте уже просмотренного объема файла. Этот ключ заставляет less вывести приглашение, аналогичное тому, что выводит утилита more. В него включается отчет о количествах

байтов, когда less ведет чтение со стандартного ввода, поскольку less не в состоянии определить размер вводимого файла.

`--LINE-NUMBERS` `-N` Вывод в начале каждой строки ее номера.

`--prompt=`*приглашение* `-P` *приглашение* Изменение строки короткого приглашения (которая появляется в нижней части каждого экрана вывода) на *приглашение*. Если *приглашение* содержит пробелы, его нужно взять в кавычки. В *приглашение* можно включить специальные символы, которые less при выводе *приглашения* заменит другими значениями. Например, вместо `%f` утилита less выводит в приглашении имя текущего файла. Перечень специальных символов и описание других вариантов приглашений приведены на man-странице. Настройка приглашений применяется при запуске less из других программ, когда нужно что-то подсказать или о чем-то проинформировать тех, кто пользуется программой. По умолчанию в приглашении выводится имя файла в негативном отображении.

`--squeeze-blank-lines` `-s` Вывод нескольких смежных пустых строк в виде одной строки. Если less используется для вывода текста, отформатированного для печати с пустыми местами в верхней и нижней части каждой страницы, этот ключ приводит к сокращению верхних и нижних отступов до одной строки.

`--tabs=`*n* `-x`*n* Установка шага табуляции равным *n* символам. По умолчанию он равен восьми символам.

`--window=`*n* `-[z]`*n* Установка шага прокрутки в *n* строк. По умолчанию используется высота отображаемого пространства в строках.

При каждом перемещении страницы вперед или назад изображение перемещается на *n* строк. Часть `z` этого ключа поддерживает совместимость с утилитой more и может быть опущена.

+команда При работе утилиты less любая команда, которая ей может быть дана, может быть также задана в командной строке в виде ключа путем установки перед командой знака плюс (+). См. раздел «Команды». Команда, перед которой в командной строке стоит знак плюс, выполняется, как только less начнет работу, и применяется только для первого файла.

++ команда Этот ключ похож на ключ *+команда*, за исключением того, что эта *команда* применяется к каждому файлу из ***списка_файлов***, а не только к первому файлу.

Примечания

Фраза «less в смысле more» объясняет происхождение имени этой утилиты. Утилита more является в Berkeley UNIX исходным средством разбиения на страницы (которое также доступно и при работе под управлением Linux). Утилита less похожа на утилиту more, но включает в себя множество усовершенствований. (При работе под управлением OS X less и more являются копиями одного и того же файла.) После отображения экрана текста less выводит приглашение и ждет ввода команды. Можно пролистать текст файла вперед и назад, вызвать текстовый редактор, провести поиск по шаблону или выполнить ряд других задач.

Способ редактирования файлов, просматриваемых с помощью утилиты less, дан в описании команды v в следующем разделе.

Ключи для less можно указать либо в командной строке при вызове этой утилиты, либо путем присваивания значения переменной среды окружения LESS. К примеру, чтобы использовать less с ключами -x4 и -s, из оболочки bash можно ввести следующую команду:

```
$ export LESS="-x4 -s"
```

Обычно значение переменной LESS присваивается в файле ~/.bash_profile, если работа ведется в оболочке bash, или в файле ~/.login, если используется оболочка tcsh. Когда установлена переменная LESS, утилита less использует указанные в ней ключи при каждом вызове. Любые ключи, указанные в командной строке, отменяют соответствующие установки в переменной LESS. Значение переменной LESS используется как при вызове less из командной строки, так и при вызове этой утилиты из другой программы, например из man. Чтобы указать less в качестве системы разбиения на страницы, используемой с man и другими программами, нужно присвоить переменной среды окружения PAGER значение less. Например, при работе в bash можно добавить к файлу ~/.bash_profile следующую строку:

```
export PAGER=less
```

Команды

При каждой паузе в работе утилиты less можно ввести любую из довольно большого количества команд. В этом разделе рассматриваются некоторые, наиболее часто используемые команды. Полный список команд дан в man-странице, посвященной less. Необязательный числовой аргумент *n* за исключением особо оговоренных случаев по умолчанию имеет значение 1. После набора этих команд нажимать ВВОД не требуется.

*n*b *или* *n*CONTROL+B (backward — назад). Прокрутка назад на *n* строк. По умолчанию значение *n* равно высоте экрана в строках.

*n*d *или* *n*CONTROL+D (down — вниз). Прокрутка вперед на *n* строк. По умолчанию значение *n* равно половине высоты экрана в строках. При указании значения *n* оно становится для этой команды новым значением *n* по умолчанию.

F (forward — вперед). Прокрутка вперед. Когда достигнуто окончание ввода, эта команда ожидает нового ввода, а затем продолжает прокрутку. Эта команда позволяет использовать less таким образом, что это похоже на использование утилиты tail с ключом -f (см. соответствующий раздел), за тем лишь исключением, что less разбивает появляющийся ввод на страницы.

*n*g (go — перейти). Переход на строку номер *n*. Эта команда может не работать, если файл читается со стандартного ввода и уже произошло достаточно глубокое продвижение по содержимому файла. По умолчанию *n* равно 1.

h *или* H (help — справка). Вывод краткого описания всех доступных команд. Краткая справка выводится с использованием самой утилиты less, поскольку список команд слишком длинный.

*n*ВВОД *или n*j (jump — перепрыгнуть). Прокрутка вперед на *n* строк. По умолчанию *n* равно 1.

q *или* :q Завершение работы less.

*n*u *или* *n*CONTROL+U Прокрутка назад на *n* строк. По умолчанию значение *n* равно половине высоты экрана в строках. При указании значения *n* оно становится новым значением по умолчанию для этой команды.

v Перенос текущего файла в редактор с установкой курсора на текущую строку. Утилита less использует редактор, указанный в переменной среды окружения EDITOR. Если значение для EDITOR не установлено, less использует vi (редактор, который обычно является ссылкой на vim).

*n*w Прокрутка назад, как и при вызове команды *n*b, за исключением того, что значение *n* становится новым значением по умолчанию для этой команды.

*n*y *или n*k Прокрутка назад на *n* строк. Значение *n* по умолчанию равно 1.

*n*z Вывод следующих *n* строк как при вводе команды *n*ПРОБЕЛ, за исключением того, что значение *n*, если оно присутствует, становится новым значением по умолчанию для команд z и ПРОБЕЛ.

*n*ПРОБЕЛ Вывод следующих *n* строк. Нажатие одного ПРОБЕЛА приводит к отображению следующего экрана текста.

/регулярное_выражение Перемещение вперед по файлу в поиске строки, в которой содержится соответствие *регулярному_выражению*. Если *регулярное_выражение* начинается с восклицательного знака (!), эта команда ищет строки, которые не содержат соответствия *регулярному_выражению*. Если *регулярное_выражение* начинается со звездочки (*), эта команда продолжает поиск по всему ***списку_файлов***. (Обычно поиск останавливается в конце текущего файла.) Если *регулярное_выражение* начинается со знака at (@), эта команда начинает поиск с начала ***списка_файлов*** и продолжает поиск до конца этого списка.

? *регулярное_выражение* Эта команда похожа на предыдущую, но ведет поиск в обратном направлении по всему файлу (и ***списку_файлов***). Звездочка (*) в качестве первого символа регулярного выражения заставляет вести поиск, продолжающийся в ***списке_файлов*** в обратном направлении, до начала первого файла. Знак at (@) заставляет вести поиск, начиная с последней строки последнего файла из ***списка_файлов*** по направлению к первой строке первого файла.

{ *или* (*или* [Если один из этих символов появляется в отображаемой на экране верхней строке, эта команда осуществляет прокрутку вперед к соответствующей правой фигурной, круглой или квадратной скобке. Например, ввод символа { заставляет утилиту less переместить курсор вперед к соответствующей скобке }.

} *или*) *или*] Эта команда аналогична предыдущей, но перемещает курсор назад, на соответствующую левую фигурную, круглую или квадратную скобку.

CONTROL+L Перерисовка экрана. Эта команда применяется, если текст на экране приобрел искаженный вид.

[*n*]:*n* Переход к следующему файлу в ***списке_файлов***. Если задан *n*, переход на *n*-й следующий файл в ***списке_файлов***.

![*командная_строка*] Выполнение командной строки под управлением оболочки, указанной в переменной среды окружения SHELL, или под управлением sh (которая обычно ссылается на bash или является ее копией) по умолчанию. Знак процента (%) в командной строке заменяется именем текущего файла. Если командная строка будет опущена, less запускает интерактивную оболочку.

Примеры

Следующий пример выводит файл memo.txt. Чтобы просмотреть следующую часть файла, пользователь в ответ на приглашение утилиты less, появляющееся в левом нижнем углу экрана, нажимает ПРОБЕЛ:

```
$ less memo.txt
…
memo.txt ПРОБЕЛ
…
```

В следующем примере пользователь изменяет приглашение на более подходящее по смыслу сообщение и использует ключ -N для вывода номеров строк. В завершение пользователь дает less команду переместиться вперед к первой строке, содержащей строку procedure.

```
$ less -Ps"Нажмите ПРОБЕЛ для продолжения, q для выхода" -N +/procedure ncut.icn
28  procedure main(args)
29      local filelist, arg, fields, delim
30
31      filelist:=[]
…
45      # Check for real field list
46      #
47      if /fields then stop("-fFIELD_LIST is required.")
48
49      # Process the files and output the fields
Нажмите ПРОБЕЛ для продолжения, q для выхода
```

ln

Создание ссылки на файл.

ln [ключи] существующий_файл [новая_ссылка]
ln [ключи] список_существующих_файлов каталог

Утилита ln создает жесткие или символьные ссылки на один или несколько файлов. Жесткая ссылка на каталог не создается, на него можно создать только символьную ссылку.

Аргументы

В первом формате ***существующий_файл*** — это имя файла, на который нужно создать ссылку. ***Новая_ссылка*** — это полное имя новой ссылки. Когда создается символьная ссылка, ***существующий_файл*** может быть каталогом. Если ***новая_ссылка*** опущена,

ln создает ссылку на ***существующий_файл*** в рабочем каталоге и использует для нее то же самое простое имя файла, которое принадлежит ***существующему_файлу***.

Во втором формате ***список_существующих_файлов*** — это список путевых имен обычных файлов, на которые нужно создать ссылки. Утилита ln создаст новые ссылки в ***каталоге***. Простые имена файлов в записях ***каталога*** будут такими же, как и простые имена файлов в ***списке_существующих_файлов***.

Ключи

Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми стоит одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--backup -b` Если утилита ln будет удалять файл, этот ключ приводит к созданию резервной копии путем добавления знака тильды (~) к имени файла. Этот ключ работает только с ключом `--force`. (Linux)

`--force -f` Обычно ln не создает ссылку, если ***новая_ссылка*** уже существует. Этот ключ перед созданием ссылки удаляет ***новую_ссылку***. Если ключи `--force` и `--backup` используются вместе (только при работе в Linux), ln создает копию ***новой_ссылки*** перед ее удалением.

`--interactive -i` Если ***новая_ссылка*** уже существует, то этот ключ перед удалением ***новой_ссылки*** выводит приглашение. Если в ответ будет введена буква `y` или слово `yes`, ln перед созданием ссылки удаляет ***новую_ссылку***. Если в ответ будет введена буква `n` или слово `no`, ln не удаляет ***новую_ссылку*** и не создает никакой новой ссылки.

`--symbolic -s` Создание символьной ссылки. При использовании этого ключа ***существующий_файл*** и ***новая_ссылка*** могут быть каталогами и могут находиться в разных файловых системах. См. раздел «Символьные ссылки» на с. 130.

Примечания

Дополнительные сведения даны в разделе «Ссылки» на с. 126. Утилита ls с ключом `-l` выводит количество жестких ссылок на файл (см. рис. 4.9).

Жесткие ссылки По умолчанию ln создает жесткие ссылки. Жесткая ссылка на файл невидима из исходного файла. Все жесткие ссылки на файл должны быть в одной файловой системе. Дополнительные сведения даны в разделе «Жесткие ссылки» на с. 128.

Символьные ссылки Утилита ln может также использоваться для создания символьных ссылок. В отличие от жесткой ссылки символьная ссылка может существовать в другой файловой системе по отношению к тому файлу, на который она ссылается. Кроме этого символьная ссылка может указывать на каталог. Дополнительные сведения даны в разделе «Символьные ссылки» на с. 130.

Если ***новая_ссылка*** является именем существующего файла, ln не будет создавать ссылку, пока не будет использован ключ `--force option` (только при работе в Linux) или пока не будет получен положительный ответ при использовании ключа `-i` (`--interactive`).

Примеры

Следующая команда создает ссылку между файлом memo2 в подкаталоге literature личного каталога пользователя Zach и рабочим каталогом. Файл появляется в рабочем каталоге под именем memo2 (под простым именем существующего файла):

```
$ ln ~zach/literature/memo2 .
```

Точку, представляющую в предыдущей команде рабочий каталог, можно опустить. Если ln получает только один аргумент, она создает ссылку в рабочем каталоге.

Следующая команда создает ссылку на тот же самый файл. На этот раз файл появляется в рабочем каталоге под именем new_memo:

```
$ ln ~zach/literature/memo2 new_memo
```

Следующая команда создает ссылку, заставляя файл появиться в личном каталоге пользователя Sam:

```
$ ln ~zach/literature/memo2 ~sam/new_memo
```

Чтобы эта команда работала, нужно иметь права на запись и выполнение в отношении каталога другого пользователя. Чтобы предоставить другому пользователю право на запись в файл, владелец файла может использовать утилиту chmod.

Следующая команда создает символьную ссылку на каталог. Команда `ls -ld` показывает эту ссылку:

```
$ ln -s /usr/local/bin bin
$ ls -ld bin
lrwxrwxrwx  1 zach zach 14 Feb 10 13:26 bin -> /usr/local/bin
```

В последнем примере предпринимается попытка создания символьной ссылки по имени memo1 на файл memo2. Поскольку файл memo1 уже существует, ln отказывается создавать ссылку. При использовании ключа `-i` (`--interactive`) ln задает вопрос о том, нужно ли удалять существующий файл memo1 с символьной ссылкой. Если ввести букву y или слово yes, ln создает ссылку и старый файл memo1 исчезает.

```
$ ls -l memo?
-rw-rw-r--   1 zach group 224 Jul 31 14:48 memo1
-rw-rw-r--   1 zach group 753 Jul 31 14:49 memo2
$ ln -s memo2 memo1
ln: memo1: File exists          ln: memo1: Файл существует
$ ln -si memo2 memo1
ln: replace 'memo1'? y        ln: заменить 'memo1'? да
$ ls -l memo?
lrwxrwxrwx   1 zach group 5 Jul 31 14:49 memo1 -> memo2
-rw-rw-r--   1 zach group 753 Jul 31 14:49 memo2
```

При работе под управлением Linux, чтобы заставить ln переписать файл, можно также воспользоваться ключом `--force`.

lpr

Отправка файлов на принтер.

lpr [ключи] [список_файлов]
lpq [ключи] [идентификаторы_заданий]
lprm [ключи] [идентификаторы_заданий]

Утилита lpr ставить один или несколько файлов в очередь на печать, предоставляя поочередный доступ к принтерам нескольким пользователям или процессам. Эта утилита может работать с принтерами, подключенными к удаленным системам. Утилита lprm может использоваться для удаления файлов из очереди на печать, а утилита lpq — для проверки состояния файлов в очереди. См. далее подраздел «Примечания».

Аргументы

Список_файлов — одно или несколько имен файлов, передаваемых lpr на печать. Зачастую это текстовые файлы, но многие системы настроены так, что lpr может принимать и успешно распечатывать файлы многих других типов, включая PostScript- и PDF-файлы. Если ***список_файлов*** не указывается, lpr принимает ввод со стандартного ввода.

Идентификаторы_заданий — это список номеров заданий или имен пользователей. Если номер задания неизвестен, то для вывода списка заданий на печать используется утилита lpq.

Ключи

Некоторые из следующих ключей зависят от типа файла, выводимого на печать, а также от того, как система настроена на печать.

`-h` (no header — без заголовка). Подавление печати заголовочной (пакетной) страницы. Эта страница применяется для идентификации владельца выводимой на печать информации в многопользовательских установках, но когда такая идентификация не нужна, это приводит к напрасной трате бумаги.

`-l` (буква «l» в нижнем регистре). Указание на то, что lpr не должна проводить предварительную обработку (фильтрацию) выводимого на печать файла. Этот ключ используется, когда файл уже отформатирован для вывода на печать.

`-P` *принтер* Направление заданий на печать в очередь для принтера по имени ***принтер***. Если этот ключ не используется, задания на печать направляются на принтер, используемый на локальной системе по умолчанию. Подходящие значения для аргумента ***принтер*** можно найти в Linux-файле /etc/printcap и можно вывести с помощью команды `lpstat -t`. Эти значения для разных систем разные.

`-r` (remove — удалить). Удаление файлов, указанных в ***списке_файлов*** после вызова утилиты lpr.

`-#` *n* Вывод на печать *n* копий каждого файла. В зависимости от типа используемой оболочки может потребоваться отключить специальное значение символа #, поставив перед ним символ обратного слэша, чтобы оболочка не интерпретировала его в этом качестве.

Комментарии

Утилита lpr получает ввод либо из файлов, указанных в командной строке, либо со стандартного ввода. Она добавляет полученные файлы в очередь на печать в виде заданий. Каждому заданию на печать утилита присваивает уникальный идентифика-

ционный номер. Утилита lpq выводит номера заданий на печать, которые были установлены с помощью утилиты lpr. Для удаления задания на печать из очереди можно воспользоваться утилитой lprm.

lpq Утилита lpq выводит информацию о заданиях, имеющихся в очереди на печать. При вызове без аргументов lpq выводит список всех заданий печати, поставленных в очередь принтера, используемого по умолчанию. Для просмотра других очередей на печать, даже для тех принтеров, которые подключены к удаленным системам, следует воспользоваться при вызове утилиты lpq ключом `-P` утилиты lpr. С ключом `-l` утилита lpq выводит о каждом задании более подробную информацию. Если в качестве аргумента предоставить имя пользователя, lpq выведет только те задания на печать, которые принадлежат этому пользователю.

lprm Одним из элементов, отображаемых утилитой lpq, является номер каждого задания на печать, стоящего в очереди. Чтобы удалить задание из очереди, нужно дать его номер в качестве аргумента утилите lprm. Работая без root-прав, можно удалять только свои собственные задания. Но даже у пользователя, работающего с root-правами, может не получиться удаление задания из очереди для удаленного принтера. Если не дать утилите lprm ни одного аргумента, она удалит из очереди активное задание на печать (то есть задание, которое сейчас выводится на печать), если вы являетесь владельцем этого задания.

Примечания

Если обычно используется принтер, который не является для системы принтером по умолчанию, можно перенастроить lpr на использование другого принтера, являющегося вашим персональным принтером по умолчанию, присвоив имя этого принтера переменной среды окружения `PRINTER`. Например, если используется оболочка bash, то для установки принтера по имени ps в качестве вашего принтера по умолчанию к файлу ~/.bash_profile можно добавить следующую строку:

```
export PRINTER=ps
```

LPD и LPR Традиционно у UNIX имелись две системы печати: демон линейного принтера BSD Line Printer Daemon (LPD) и система линейного принтера System V Line Printer system (LPR). Сначала Linux перенял эти системы, а затем и в UNIX, и в Linux появились модификации этих систем и средства, предназначенные для их замены. Сейчас системой вывода на печать как в Linux, так и в OS X является CUPS.

CUPS CUPS представляет собой кросс-платформенный сервер печати, созданный на основе протокола печати Интернета — Internet Printing Protocol (IPP), который, в свою очередь, основан на протоколе HTTP. CUPS предоставляет большое количество драйверов принтеров и может выводить на печать файлы различных типов, включая PostScript-файлы. CUPS предоставляет интерфейсы командных строк System V и BSD и в дополнение к ним IPP, поддерживающий, кроме всех прочих, протоколы LPD/LPR, HTTP, SMB и JetDirect (сокет).

В этом разделе рассматривается интерфейс командной строки LPD, который запускается под управлением CUPS, а также в своем собственном режиме на устаревших системах.

Примеры

Первая команда отправляет файл по имени memo2 на принтер, используемый по умолчанию:

```
$ lpr memo2
```

Затем канал отправляет вывод команды ls на принтер по имени deskjet:

```
$ ls | lpr -Pdeskjet
```

В следующем примере разбивается на страницы и отправляется на принтер файл memo:

```
$ pr -h "Today's memo" memo | lpr
```

В следующем примере демонстрируется количество заданий на печать, стоящих в очереди для принтера, используемого по умолчанию. Все задания принадлежат пользователю Max, и первое из них выводится на печать (является активным). Задания 635 и 639 были созданы путем отправки ввода на стандартный ввод lpr; задание 638 было создано путем передачи имени файла ncut.icn в качестве аргумента команде lpr. В последнем столбце дается размер каждого задания на печать.

```
$ lpq
deskjet is ready and printing
Rank    Owner     Job  Files                                 Total Size
active  max       635  (stdin)                              38128 bytes
1st     max       638  ncut.icn                              3587 bytes
2nd     max       639  (stdin)                               3960 bytes
```

В следующей команде задание на печать 638 удаляется из очереди принтера, используемого по умолчанию:

```
$ lprm 638
```

ls

Вывод информации об одном или нескольких файлах.

ls [ключи] [Список_файлов]

Утилита ls выводит информацию об одном или нескольких файлах. Если не используется ключ, изменяющий порядок вывода, информационный список выводится в алфавитном порядке имен файлов.

Аргументы

Если аргумент не предоставляется, ls выводит имена отображаемых файлов (чьи имена не начинаются с точки) в рабочем каталоге. ***Список_файлов*** — это список, состоящий из одного или нескольких путевых имен любых обычных файлов, файлов каталогов и файлов устройств. В него могут включаться нестрогие ссылки на имена файлов. Когда ***список_файлов*** включает каталог, утилита ls выводит содержимое каталога. Она выводит имя каталога, только если нужно избежать неоднозначности, например когда в списке более одного каталога. Если указать обычный файл, ls выводит информацию только об этом файле.

Ключи

Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

Ключи определяют тип выводимой утилитой ls информации, способ и порядок ее выведения. Если ключ не используется, ls выводит краткий список, содержащий только имена файлов в алфавитном порядке.

`--almost-all` `-A` Ключ, аналогичный ключу `-a`, но не вызывающий включение в список записей каталогов . и ..

`--all` `-a` Включение в список имен информации о файлах со скрытыми именами (которые начинаются с точки, см. раздел «Скрытые имена файлов» на с. 102). Без этого ключа ls не выводит информацию о скрытых файлах, пока их имена не будут включены в ***список_файлов***. Нестрогая ссылка на имена файлов * не соответствует начальной точке в имени файла (см. раздел «Специальный символ „*“» на с. 158), поэтому для вывода информации о скрытых файлах нужно воспользоваться этим ключом или в явном виде указать имя файла (в виде нестрогой ссылки или в виде самого имени).

`--escape` `-b` Вывод в именах файлов невыводимых на печать символов с использованием эскейп-последовательностей, образуемых с помощью обратного слэша, подобных тем, которые используются в строках языка C (табл. V.18). Другие невыводимые на печать символы отображаются в виде обратного слэша, за которым следует восьмеричное число.

Таблица V.18. Эскейп-последовательности, образуемые с помощью обратного слэша

Последовательность	Значение
\b	BACKSPACE
\n	Разделитель строк (NEWLINE)
\r	Ввод (RETURN)
\t	Символ горизонтальной табуляции (HORIZONTAL TAB)
\v	Символ вертикальной табуляции (VERTICAL TAB)
\\	Обратный слэш (BACKSLASH)

`--color` `[=`*когда*`]` Утилита ls может выводить информацию о разных типах файлов в разных цветах, но обычно цвета не используются (такой же результат получается при указании в качестве аргумента *когда* слова, означающего никогда – `none`). Если аргумент *когда* не указан или если в качестве его значения используется слово «всегда»(`always`), ls использует цветовое оформление. Если в качестве значения *когда* указать слово «автоматически» (`auto`), ls использует цвета только в том случае, когда вывод идет на экран. Дополнительные сведения даны в подразделе «Примечания». (Linux)

`--directory` `-d` Вывод каталогов без отображения их содержимого. Этот ключ не разыменовывает символьные ссылки; то есть для каждого файла, являющегося символьной ссылкой, этот ключ приводит к выводу информацию о символьной ссылке, а не о файле, на который она указывает.

`-e` Вывод ACL-списков. (OS X)

`--format=`*слово* По умолчанию ls выводит информацию о файлах, отсортированную по вертикали. Этот ключ приводит к сортировке информации о файлах, исходя из содержимого *слова*: поперек или по горизонтали (точно так же, как `-x`), с запятыми в качестве разделителей (точно так же, как `-m`), в длину (точно так же, как `-l`) или в один столбец (точно так же, как `-1`). (Linux)

`--classify -F` Вывод слэша (/) после каждого каталога, звездочки (*) после каждого исполняемого файла и знака at (@) после каждой символьной ссылки.

`--dereference-command-line -H` (частичное разыменование). Для каждого файла, являющегося символьной ссылкой, вывод информации о файле, на который указывает ссылка, а не о самой символьной ссылке. Действие этого ключа распространяется на файлы, указанные в командной строке, оно не касается файлов, найденных при спуске по иерархии каталога. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример использования этого ключа приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L».

`--human-readable -h` При использовании вместе с ключом `-l` вывод размеров там, где это приемлемо, в K (килобайтных), M (мегабайтных) и G (гигабайтных) блоках. Этот ключ работает только с ключом `-l`. Использует степени числа 1024. При работе под управлением Mac OS X, вдобавок к предыдущим суффиксам используется также B (байт). См. также описание ключа `--si`.

`--inode -i` Вывод номера i-узла каждого файла. При использовании с ключом `-l` этот ключ выводит номер i-узла в столбце 1, сдвигая все остальные элементы на один столбец вправо.

`--dereference -L` (разыменование). Для каждого файла, являющегося символьной ссылкой, вывод информации о файле, на который указывает ссылка, а не о самой символьной ссылке. Действие этого ключа распространяется на все файлы. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример использования этого ключа приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L».

`--format=long -l` (буква «l» в нижнем регистре). Вывод дополнительной информации о каждом файле. Использование этого ключа не приводит к разыменованию символьных ссылок, то есть для каждого файла, являющегося символьной ссылкой, этот ключ приводит к выводу информации о символьной ссылке, а не о файле, на которую она указывает. Если стандартный вывод информации о каталогах направлен на экран, использование этого ключа приводит к выводу количества блоков, использованных всеми файлами, которое указывается на строке, выводимой перед списком. Чтобы размеры файлов легче читались, этот ключ следует использовать вместе с ключом `-h`. Дополнительные сведения даны в разделе «Комментарии».

`--format=commas -m` Вывод списка имен файлов, разделенных запятыми, во всю ширину экрана.

`-P` (без разыменования). Для каждого файла, являющегося символьной ссылкой, вывод информации о символьной ссылке, а не о файле, на который она

указывает. Действие этого ключа распространяется на все файлы. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример использования этого ключа приведен в разделе «chgrp», в подразделе «Примеры». (OS X)

`--hide-control-chars` `-q` — Вывод невыводимых на печать символов в имени файла в виде вопросительных знаков. Когда стандартный вывод направлен на экран, утилита делает это по умолчанию. Без этого ключа, когда стандартный вывод отправляется в фильтр или в файл, невыводимые на печать символы выводятся как есть.

`--recursive` `-R` — Рекурсивный вывод списка иерархий каталогов.

`--reverse -r` — Вывод списка имен файлов в обратном отсортированном порядке.

`--size -s` — Вывод количества 1024-байтных (Linux) или 512-байтных (Mac OS X) блоков, выделенных файлу. Размер указывается перед именем файла. При использовании ключа `-l` этот ключ приводит к отображению размера в столбце 1 и к сдвигу других элементов на один столбец вправо. Если стандартный вывод информации о каталогах направлен на экран, использование этого ключа приводит к выводу количества блоков, использованных всеми файлами, которое указывается на строке, выводимой перед списком. Чтобы размеры файлов легче читались, этот ключ следует использовать вместе с ключом `-h`. При работе под управлением Mac OS X для изменения размера блоков, показываемого при использовании этого ключа, можно воспользоваться переменной среды окружения `BLOCKSIZE` (см. ее описание в начале части V).

`--si` — При использовании вместе с ключом `-l` вывод размеров там, где это приемлемо, в K (килобайтных), M (мегабайтных) и G (гигабайтных) блоках. Этот ключ работает только с ключом `-l`. Использует степени числа 1000. См. также описание ключа `--human-readable`. (Linux)

`--sort=time` `-t` — Вывод информации о файлах, отсортированной по времени внесения в них последних изменений.

`--sort=`*слово* — По умолчанию ls выводит файлы в порядке, обуславливаемом ASCII-кодами символов, используемых в их именах. Этот ключ осуществляет сортировку информации о файлах исходя из содержимого *слова*: по расширениям имен файлов (`-X`; только в Linux), без сортировки (`-U`; только в Linux), по размерам файлов (`-S`), по времени последнего обращения (`-u`) или по времени внесения последних изменений (`-t`). Исключения рассмотрены в описании ключа `--time`. (Linux)

`--time=`*слово* — По умолчанию ls с ключом `-l` выводит время последнего изменения файла. Чтобы вывести время последнего обращения к файлу в качестве значения *слова*, нужно использовать `atime` (`-u`), а чтобы вывести время внесения последних изменений, нужно использовать `ctime` (`-t`). Список сортируется по *слову*, когда также будет указан ключ `--sort=time`. (Linux)

`--sort=access` `-u` — Вывод информации о файлах, отсортированной по времени последнего обращения к этим файлам.

`--format=extension -X` Вывод информации о файлах, отсортированной по расширениям имен файлов. Сначала в списке выводится информация о файлах, чьи имена не имеют расширений. (Linux)

`--format=across -x` Вывод информации о файлах, отсортированной по строкам (по умолчанию она сортируется по столбцам).

`--format=single-column -1` (единица). Вывод в каждой строке информации только об одном файле. Если вывод утилиты ls перенаправляется, этот тип вывода используется по умолчанию.

Комментарии

В подробном списке ls (при вызове с ключами `-l` или `--format=long`) выводятся столбцы, показанные на рис. 4.9. Первый столбец, содержащий 10 или 11 символов, поделен на части, описываемые в следующих абзацах. Символ в первой позиции описывает тип файла, как показано в табл. V.19.

Таблица V.19. Первый символ в подробном списке, выводимом утилитой ls

Символ	Значение
-	Обычный
B	Блочное устройство
C	Символьное устройство
D	Каталог
P	FIFO (именованный канал)
L	Символьная ссылка

Следующие девять символов первого столбца описывают права доступа, связанные с файлом. Эти символы разбиты на три набора по три символа в каждом.

Первые три символа представляют права доступа, имеющиеся у владельца файла. Если у владельца есть право на чтение файла, то в первой символьной позиции стоит символ `r`. Если у владельца нет права на чтение файла, в этой позиции стоит дефис. Следующие две позиции описывают права владельца на запись и выполнение. Если во второй позиции стоит символ w, владелец имеет право записи в файл, а если в третьей позиции стоит символ `x`, то владелец имеет право на выполнение файла. Символ `s` в третьей позиции свидетельствует о том, что для файла установлен бит `setuid`, а также что его владелец имеет право на выполнение. Дефис показывает, что владелец не имеет права доступа, связанного с данной символьной позицией.

Точно так же второй набор из трех символов представляет права доступа для группы, с которой связан файл. Символ `s` в третьей позиции свидетельствует о том, что для файла установлен бит `setgid`, а также право на выполнение, а символ `S` свидетельствует о том, что бит `setgid` установлен, но право на выполнение отсутствует.

Третий набор из трех символов описывает права доступа для всех остальных пользователей. Символ `t` в третьей позиции свидетельствует о том, что этот файл имеет установленный бит флага ограничения прав на удаление (sticky bit).

Сведения об изменении прав доступа даны в описании утилиты chmod.

Если разрешено использование ACL-списков (см. раздел «Списки управления доступом (ACL-списки)» на с. 121) и перечисляемый файл имеет ACL-список, команда `ls -l` приводит к выводу знак плюс (+), который следует за третьим набором из трех символов.

Если еще раз взглянуть на рис. 4.9, то второй столбец показывает количество жестких ссылок на файл. Дополнительные сведения о ссылках даны в разделе «Ссылки» на с. 126.

В третьем и четвертом столбцах выводятся соответственно имя владельца файла и имя группы, с которой этот файл связан.

Пятый столбец показывает размер файла в байтах или, если отображается информация о файле устройства, старший и младший номер устройства. Если отображается информация о каталоге, в этом столбце показывается размер файла каталога, но не размер файлов, записи о которых находятся внутри каталога. (Чтобы вывести суммарный размер всех файлов, находящихся в каталоге, нужно воспользоваться утилитой du.) Чтобы вывести размер файлов в килобайтах, мегабайтах или гигабайтах, нужно воспользоваться ключом `-h`.

В последних двух столбцах выводится соответственно дата и время последнего изменения файла и его имя.

Примечания

По умолчанию ls разыменовывает символьные ссылки: для каждого файла, являющегося символьной ссылкой, ls выводит информацию о файле, на который указывает ссылка, а не о самой символьной ссылке. Для обращения к символьным ссылкам (вывода информации о самих символьных ссылках, а не о файлах, на которые они указывают) нужно использовать ключ `-l` или ключ `-d`.

При выводе информации не в подробном режиме (без ключа `-l`), когда стандартный вывод направляется на экран, ls отображает вывод в столбцах, основываясь на ширине экрана. Если стандартный вывод перенаправляется в фильтр или файл, ls выводит один столбец.

Примеры использования ls с нестрогими ссылками на имена файлов даны в разделе «Генерация имени файла и расширение имени» на с. 157.

При использовании ключа `--color` ls выводит имена файлов разных типов в разных цветах. По умолчанию имена исполняемых файлов выводятся в зеленом цвете, каталогов — в синем, символьных ссылок — в голубом, архивов и сжатых файлов — в красном и обычных файлов — в черном цвете. Система выделения цветом имен файлов разного типа указана в файле /etc/DIR_COLORS. Если этот файл на локальной системе отсутствует, ls не будет выделять имена файлов цветом. Чтобы изменить используемую по умолчанию карту соотношения цветов с типами файлов на всей системе, можно отредактировать файл /etc/DIR_COLORS. Для личного пользования можно скопировать файл /etc/DIR_COLORS в файл ~/.dir_colors в своем личном каталоге и внести изменения в этот файл. При входе в систему файл ~/.dir_colors отменяет цветовые установки для всей системы, находящиеся в файле /etc/DIR_COLORS. Дополнительные сведения даны на man-страницах `dir_colors` и `dircolors`.

Примеры

В первом примере показано, что без ключей и аргументов ls выводит отсортированный в алфавитном порядке список информации о файлах, имеющихся в рабочем каталоге. Список сортируется по столбцам (вертикально):

```
$ ls
bin  calendar  letters
c    execute   shell
```

В следующем примере показана утилита ls с ключом -x, который приводит к сортировке файлов по горизонтали:

```
$ ls -x
bin      c        calendar
execute  letters  shell
```

Ключ -F приводит к добавлению слэша (/) к файлам, являющимся каталогами, звездочки (*) к исполняемым файлам и знака at (@) к файлам, являющимся символьными ссылками:

```
$ ls -Fx
bin/      c/        calendar
execute*  letters/  shell@
```

Следующий ключ -l (long — длинный) приводит к выводу подробного списка. Имена файлов выводятся по-прежнему в алфавитном порядке:

```
$ ls -l
total 20
drwxr-xr-x 2 sam pubs 4096 May 20 09:17 bin
drwxr-xr-x 2 sam pubs 4096 Mar 26 11:59 c
-rw-r--r-- 1 sam pubs  104 Jan  9 14:44 calendar
-rwxr-xr-x 1 sam pubs   85 May  6 08:27 execute
drwxr-xr-x 2 sam pubs 4096 Apr  4 18:56 letters
lrwxrwxrwx 1 sam sam     9 May 21 11:35 shell -> /bin/bash
```

Ключ -a (all — все) приводит к выводу информации о всех файлах, включая те, которые имеют скрытые имена:

```
$ ls -a
.        bin      execute
..       c        letters
.profile calendar shell
```

Сочетание ключей -a и -l приводит к выводу подробного списка всех файлов рабочего каталога, включая невидимые. Этот список по-прежнему выводится в алфавитном порядке:

```
$ ls -al
total 32
drwxr-xr-x 5 sam sam  4096 May 21 11:50 .
drwxrwxrwx 3 sam sam  4096 May 21 11:50 ..
-rw-r--r-- 1 sam sam   160 May 21 11:45 .profile
drwxr-xr-x 2 sam pubs 4096 May 20 09:17 bin
drwxr-xr-x 2 sam pubs 4096 Mar 26 11:59 c
-rw-r--r-- 1 sam pubs  104 Jan  9 14:44 calendar
-rwxr-xr-x 1 sam pubs   85 May  6 08:27 execute
```

```
drwxr-xr-x 2 sam pubs 4096 Apr  4 18:56 letters
lrwxrwxrwx 1 sam sam     9 May 21 11:35 shell -> /bin/bash
```

При добавлении в командную строку ключа -r (reverse — обратный) ls выдает список в обратном алфавитном порядке:

```
$ ls -ral
total 32
lrwxrwxrwx 1 sam sam     9 May 21 11:35 shell -> /bin/bash
drwxr-xr-x 2 sam pubs 4096 Apr  4 18:56 letters
-rwxr-xr-x 1 sam pubs   85 May  6 08:27 execute
-rw-r--r-- 1 sam pubs  104 Jan  9 14:44 calendar
drwxr-xr-x 2 sam pubs 4096 Mar 26 11:59 c
drwxr-xr-x 2 sam pubs 4096 May 20 09:17 bin
-rw-r--r-- 1 sam sam   160 May 21 11:45 .profile
drwxrwxrwx 3 sam sam  4096 May 21 11:50 ..
drwxr-xr-x 5 sam sam  4096 May 21 11:50 .
```

Для вывода информации о файлах в том порядке, в котором в них вносились последние изменения, следует использовать ключи -t и -l:

```
$ ls -tl
total 20
lrwxrwxrwx 1 sam sam     9 May 21 11:35 shell -> /bin/bash
drwxr-xr-x 2 sam pubs 4096 May 20 09:17 bin
-rwxr-xr-x 1 sam pubs   85 May  6 08:27 execute
drwxr-xr-x 2 sam pubs 4096 Apr  4 18:56 letters
drwxr-xr-x 2 sam pubs 4096 Mar 26 11:59 c
-rw-r--r-- 1 sam pubs  104 Jan  9 14:44 calendar
```

Использование сочетания ключей -r и -t приводит к тому, что список возглавляют файлы, изменявшиеся раньше всех остальных:

```
$ ls -trl
total 20
-rw-r--r-- 1 sam pubs  104 Jan  9 14:44 calendar
drwxr-xr-x 2 sam pubs 4096 Mar 26 11:59 c
drwxr-xr-x 2 sam pubs 4096 Apr  4 18:56 letters
-rwxr-xr-x 1 sam pubs   85 May  6 08:27 execute
drwxr-xr-x 2 sam pubs 4096 May 20 09:17 bin
lrwxrwxrwx 1 sam sam     9 May 21 11:35 shell -> /bin/bash
```

В следующем примере показана утилита ls, в качестве аргумента которой используется имя каталога. Утилита ls выводит список содержимого этого каталога в алфавитном порядке:

```
$ ls bin
c  e  lsdir
```

Чтобы вывести информацию о самом файле каталога, следует воспользоваться ключом -d (directory — каталог). Этот ключ приводит к выводу информации только о самом каталоге:

```
$ ls -dl bin
drwxr-xr-x 2 sam pubs 4096 May 20 09:17 bin
```

Следующую команду можно применить для вывода списка информации обо всех файлах со скрытыми именами (которые начинаются с точки), имеющихся в вашем

личном каталоге. Этот способ хорошо подходит для вывода информации о всех файлах запуска (инициализации), имеющихся в личном каталоге.

```
$ ls -d ~/.*
/home/sam/.
/home/sam/..
/home/sam/.AbiSuite
/home/sam/.Azureus
/home/sam/.BitTornado
...
```

Знак плюс (+) справа от прав доступа в подробном списке означает, что для этого файла имеется ACL-список:

```
$ ls -l memo
-rw-r--r--+ 1 sam pubs 19 Jul 19 21:59 memo
```

При работе под управлением Mac OS X для вывода ACL-списка можно воспользоваться ключом `-le`:

```
$ ls -le memo
-rw-r--r-- + 1 sam pubs 19 Jul 19 21:59 memo
 0: user:jenny allow read
```

make

Поддержка актуального состояния набора программ.

make [ключи] [целевые_файлы] [аргументы]

GNU-утилита make поддерживает набор исполняемых программ в актуальном состоянии, основываясь на разнице в показателях времени внесения последних изменений в программы и в исходные файлы, от которых зависит каждая программа.

Аргументы

Целевые_файлы ссылаются на цели, указанные в строках зависимостей в файле сборки (make-файле). ***Аргументы*** командной строки, использующие форму *имя=значение*, устанавливают *значение* для переменной с *именем*, определенным внутри файла сборки. Дополнительные сведения даны в подразделе «Комментарии».

Ключи

Если не используется ключ `-f`, утилита make получает ввод из файла по имени GNUmakefile, makefile или Makefile (именно в таком порядке), находящегося в рабочем каталоге. В этом разделе ссылка на данный файл ввода имеет вид makefile. Многие пользователи предпочитают использовать имя Makefile, поскольку при выводе информации о содержимом каталогов он появляется первым.

ВЕРСИЯ MAKE, ИМЕЮЩАЯСЯ В MAC OS X, ДОПУСКАЕТ ИСПОЛЬЗОВАНИЕ ДЛИННЫХ КЛЮЧЕЙ

Ключи для утилиты make, перед которыми ставится двойной дефис (– –), работают как под Mac OS X, так и под Linux.

`--directory=`*каталог* `-C` *каталог* — Изменение перед запуском каталогов на *каталог*.

`--debug -d` — Вывод информации о том, как make решает, что ей делать.

`--file=` *файл* `-f` *файл* — (файл ввода). Использование в качестве ввода *файла* вместо файла сборки.

`--jobs[=`*n*`] -j [`*n*`]` — (jobs — задания). Одновременный запуск *n* команд вместо запускаемой по умолчанию одной команды. Одновременный запуск сразу нескольких команд особенно эффективен при работе на многопроцессорной системе. Если опустить *n*, make не ограничивает количество одновременно выполняемых заданий.

`--keep-going -k` — Продолжение обработки следующего по списку файла из ***целевых_файлов*** вместо прекращения работы при сбое в команде сборки.

`--just-print` *или* `--dry-run -n` — (без выполнения). Вывод, но не выполнение команд, которые make выполняла бы для приведения ***целевых_файлов*** в актуальное состояние.

`--silent` *или* `--quiet -s` — Отказ от вывода имен выполняемых команд.

`--touch -t` — Обновление показателей времени последнего изменения ***целевых_файлов*** без выполнения каких-либо команд сборки. См. описание утилиты touch.

Комментарии

Утилита make выполняет свои действия на основе показаний времени последнего изменения программ и исходных файлов, от которых зависит каждая программа. Каждая из исполняемых программ или каждый из ***целевых_файлов*** зависит от одного или нескольких предварительных файлов. Отношения между ***целевыми_файлами*** и предварительными файлами указаны в строках описания зависимости в сборочном файле. Команды сборки, которые следуют за строкой зависимости, определяют, как именно make может обновить ***целевые_файлы***. Пример сборочных файлов дан в соответствующем подразделе.

Документация Дополнительные сведения об утилите make и о сборочных файлах даны по адресу www.gnu.org/software/make/manual/make.html, а также на info-странице, посвященной утилите make.

Хотя чаще всего make используется для сборки программ из исходного кода, у этой универсальной утилиты сборки довольно широкий диапазон применения. Везде, где можно определить набор зависимостей для перехода из одного состояния в другое, появляется возможность использования утилиты make.

Источником основной части возможностей make являются свойства, настраиваемые в файле сборки. Например, можно определить переменные, используя тот же синтаксис, который имеется в оболочке Bourne Again Shell. Оболочку, которая требуется для выполнения команд сборки, нужно всегда указывать в файле сборки в переменной SHELL, присваивая ей значение полного имени оболочки. Для определения переменной

и присваивания ей значения нужно ближе к началу файла сборки поместить следующую строку:

```
SHELL=/bin/sh
```

Присваивание значения /bin/sh переменной SHELL позволяет использовать файл сборки на других компьютерных системах. На Linux-системах /bin/sh обычно является ссылкой на /bin/bash или /bin/dash. При работе под управлением Mac OS X /bin/sh является копией bash, которая пытается эмулировать работу настоящей оболочки Bourne Shell. Если не присвоить значение переменной SHELL в файле сборки, утилита make использует значение переменной среды окружения SHELL. Если значение SHELL не является путем к той оболочке, которая намечена для использования, и если значение SHELL не установлено в файле сборки, команды сборки могут потерпеть неудачу.

С утилитой make связан дополнительный перечень свойств.

- Определенные команды сборки можно запускать в режиме молчания, поставив перед ними знак at (@). Например, следующие строки при запуске команды make help выведут краткую подсказку:

```
    help:
@echo "Можно осуществить сборку следующих программ:"
@echo " "
@echo "libbuf.a   -- библиотеки буфера"
@echo "Bufdisplay -- отображения буфера любого формата"
@echo "Buf2ppm    -- преобразования буфера в pixmap"
```

  В предыдущем примере без знака @ make будет выводить перед выполнением каждую команду echo. Этот способ вывода сообщения работает, потому что в рабочем каталоге нет файла по имени help. В результате make запускает команды сборки в попытке создать такой файл. Поскольку команды сборки выводят сообщения, а не занимаются сборкой файла help, команду make help можно запускать повторно с одним и тем же результатом.

- Утилиту make можно заставить игнорировать статус выхода из команд, поставив перед командами знак дефиса (-). Например, следующая строка позволяет утилите make продолжить работу независимо от того, будет ли успешным вызов /bin/rm (вызов /bin/rm будет неудачным, если файл libbuf.a будет отсутствовать):

```
-/bin/rm libbuf.a
```

- Для ссылки на информацию, которая может изменяться от одного запуска make к другому, можно использовать специальные переменные. Такая информация может включать файлы, требующие обновления, файлы более свежих версий, чем у целевых файлов, и файлы, соответствующие шаблону. Например, можно воспользоваться в команде сборки переменной $?, чтобы идентифицировать все предварительные файлы, которые новее, чем целевой файл. Эта переменная позволяет выводить на печать любые файлы, изменившиеся с момента их последней распечатки:

```
list: .list
.list: Makefile buf.h xtbuff_ad.h buff.c buf_print.c xtbuff.c
pr $? | lpr
date >.list
```

  Целевой файл .list зависит от исходных файлов, которые могут быть выведены на печать. Команда сборки pr $? | lpr выводит на печать только те исходные

файлы, которые новее файла .list. Строка `date > .list` изменяет файл .list, поэтому он становится новее любого исходного файла. При следующем запуске команды `make list` будут выведены на печать только измененные файлы.

- Другие файлы сборки могут включаться в текущий файл сборки, как будто они являются его составной частью. Следующая строка заставляет make читать файл Make.config и рассматривать его содержимое как часть текущего файла сборки, позволяя вам помещать информацию, общую для более чем одного файла сборки, в одно место:

```
include Make.config
```

Примечание

При работе под управлением Mac OS X утилита make является частью дополнительно устанавливаемого инструментария разработчика — Developer Tools.

Примеры

В первом примере утилита make вынуждена обновить ***целевой_файл*** по имени analysis путем запуска трех команд cc. Она использует файл сборки по имени GNUmakefile, makefile или Makefile, находящийся в рабочем каталоге.

```
$ make analysis
cc -c analy.c
cc -c stats.c
cc -o analysis analy.o stats.o
```

В следующем примере также обновляется файл analysis, но уже с использованием файла сборки по имени analysis.mk, который находится в рабочем каталоге:

```
$ make -f analysis.mk analysis
'analysis' is up to date.   'analysis' находится в актуальном состоянии.
```

В следующем примере выводится перечень команды, которые make будет выполнять, чтобы привести в актуальное состояние ***целевой_файл*** по имени credit. Благодаря использованию ключа -n (no-execution — без выполнения) make эти команды не выполняет.

```
$ make -n credit
cc -c -O credit.c
cc -c -O accounts.c
cc -c -O terms.c
cc -o credit credit.c accounts.c terms.c
```

В следующем примере для обновления времени последнего изменения ***целевого_файла*** по имени credit используется ключ -t. После использования этого ключа make считает файл credit приведенным в актуальное состояние.

```
$ make -t credit
$ make credit
'credit' is up to date.                    'credit' не требует обновления.
```

Примеры сборочных файлов

Следующий файл Makefile является очень простым примером файла сборки. В этом файле задается компиляция программы morning (целевого файла). Первая строка является строкой зависимости, показыва-

ющей, что morning зависит от morning.c. Следующая строка является строкой сборки: она показывает, как создать morning, используя C-компилятор gcc. Строка сборки должна иметь отступ, образованный не пробелами, а символом табуляции.

```
$ cat Makefile
morning: morning.c
TAB gcc -o morning morning.c
```

Когда дается команда make, утилита make компилирует morning.c, если этот файл имеет более позднее время последнего изменения, чем у файла morning.

В следующем примере показан простой файл сборки для создания утилиты по имени ff. Поскольку команда cc, необходимая для сборки ff, имеет довольно сложную форму, использование файла сборки позволяет легко создать ff заново, не запоминая и не набирая команду cc еще раз.

```
$ cat Makefile
# Создание команды ff из исходного файла fastfind.c
SHELL=/bin/sh

ff:
gcc -traditional -O2 -g -DBIG=5120 -o ff fastfind.c myClib.a

$ make ff
gcc -traditional -O2 -g -DBIG=5120 -o ff fastfind.c myClib.a
```

В следующем примере файл сборки поддерживает актуальность файла по имени compute. Утилита make игнорирует строки комментариев (которые начинаются с символа решетки #); первые три строки следующего файла сборки являются строками комментария. Первая строка зависимости показывает, что файл compute зависит от двух объектных файлов: compute.o и calc.o. Соответствующая строка сборки дает команде make все необходимое для создания файла compute. Вторая строка зависимости показывает, что файл compute.o зависит не только от своего исходного файла на языке C, но также и от файла заголовка compute.h. Строка сборки для compute.o использует оптимизатор C-компилятора (ключ -O3). Третий набор строки зависимости и строки сборки не является обязательным. Если он отсутствует, утилита make предполагает, что calc.o зависит от calc.c, и создает строку с командой, необходимой для компиляции.

```
$ cat Makefile
#
# Файл сборки для compute
#
compute: compute.o calc.o
gcc -o compute compute.o calc.o

compute.o: compute.c compute.h
gcc -c -O3 compute.c

calc.o: calc.c
gcc -c calc.c

clean:
rm *.o *core* *~
```

Для последнего целевого файла clean предварительных файлов не существует. Эта цель часто используется для удаления уже ненужных или устаревших файлов, например файлов с расширением .o.

В следующем примере показан более сложный файл сборки, использующий свойства, не рассматривавшиеся в данном разделе. Сведения об этих и других расширенных свойствах приведены в информационных источниках, перечисленных выше в подразделе «Документация».

```
$ cat Makefile
#########################################################
## сборка и поддержка библиотеки буфера
#########################################################
SHELL=/bin/sh

#########################################################
## Флаги и библиотеки для компиляции. XLDLIBS нужна
#   при каждой сборке программы, использующей библиотеку.
#   CCFLAGS предоставляет максимальную оптимизацию.
CC=gcc
CCFLAGS=-O2 $(CFLAGS)
XLDLIBS= -lXaw3d -lXt -lXmu -lXext -lX11 -lm
BUFLIB=libbuf.a

#########################################################
## Прочее
INCLUDES=buf.h
XINCLUDES=xtbuff_ad.h
OBJS=buff.o buf_print.o xtbuff.o

#########################################################
## Здесь 'make' просто создает подсказку
help: Help
        @echo "Можно осуществить сборку следующих программ:"
        @echo " "
        @echo " libbuf.a         -- библиотеки буфера"
        @echo " bufdisplay       -- отображения буфера любого формата"
        @echo " buf2ppm          -- преобразование буфера в pixmap"

#########################################################
## Главная цель – библиотека
libbuf.a: $(OBJS)
    -/bin/rm libbuf.a

    ar rv libbuf.a $(OBJS)
    ranlib libbuf.a

#########################################################
## Второстепенные цели – утилиты, собираемые из библиотеки
bufdisplay: bufdisplay.c libbuf.a
    $(CC) $(CCFLAGS) bufdisplay.c -o bufdisplay $(BUFLIB) $(XLDLIBS)
```

```
buf2ppm: buf2ppm.c libbuf.a
    $(CC) $(CCFLAGS) buf2ppm.c -o buf2ppm $(BUFLIB)

#######################################################
## Сборка отдельных объектных модулей
buff.o: $(INCLUDES) buff.c
    $(CC) -c $(CCFLAGS) buff.c

buf_print.o:$(INCLUDES) buf_print.c
    $(CC) -c $(CCFLAGS) buf_print.c

xtbuff.o: $(INCLUDES) $(XINCLUDES) xtbuff.c
    $(CC) -c $(CCFLAGS) xtbuff.c
```

Утилита make может использоваться для задач, не связанных с компиляцией кода. В последнем примере предполагается наличие базы данных, в которой в двух столбцах перечисляются IP-адреса и соответствующие имена хостов, база данных выводит эти значения в файл по имени namedhosts.tab. Вам из этого файла нужно лишь извлечь имена хостов и создать веб-страницу hosts.html, содержащую эти имена. Следующий файл сборки является простым генератором отчета:

```
$ cat makefile
#
SHELL=/bin/bash
#
hosts.html: hosts.tab
        @echo "<HTML><BODY>" > hosts.html
        @awk '{print $$2, "<br>"}' hosts.tab >> hosts.html
        @echo "</BODY></HTML>" >> hosts.html
```

man

Вывод документации по командам.

man [ключи] [раздел] команда
man –k ключевое_слово

Утилита man (manual — руководство) предоставляет онлайн-документацию для команд Linux и Mac OS X. Кроме команд, доступных пользователю, документация охватывает множество других команд и деталей, имеющих отношение к Linux и OS X. Поскольку многие команды Linux и OS X были разработаны в рамках проекта GNU, утилита этого же проекта info (см. раздел «info: Отображение информации об утилитах» на с. 56) зачастую предоставляет более полную информацию об этих командах.

С каждой страницей руководства связан однострочный заголовок, который состоит из имени команды, названия раздела руководства, в котором находится команда, и краткого описания предназначения команды. Этот заголовок хранится в базе данных, позволяя осуществлять быстрые поисковые операции по ключевым словам, связанным с каждой man-страницей.

Аргументы

Аргумент ***раздел*** заставляет man ограничить поиск конкретным разделом руководства (см. раздел «man: Отображение системного руководства», подраздел «Разделы руководства» на с. 54). Без этого аргумента man ведет поиск в разделах в порядке их нумерации и выводит первую же найденную man-страницу. Второй формат команды man, в котором используется ключ `-k`, позволяет искать ***ключевое_слово*** в базе данных, содержащей заголовки man-страниц; man выводит список заголовков, в которых содержится ***ключевое_слово***. Команда `man -k` выполняет те же функции, что и утилита apropos (см. раздел «apropos: Поиск по ключевому слову» на с. 55).

Ключи

Ключи, перед которыми ставится двойной дефис (`--`), работают только под управлением Linux. Но среди них встречаются и такие ключи, которые работают не во всех дистрибутивах Linux. Однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--all -a` Вывод man-страниц для всех разделов руководства. Без этого ключа man выводит только первую найденную страницу. Этот ключ используется, когда неизвестно, в каком именно разделе содержится нужная информация.

`-K` *ключевое_слово* Поиск *ключевого_слова* по всем man-страницам. Выполнение операции, задаваемой этим ключом, может занять довольно много времени. В некоторых дистрибутивах Linux этот ключ не работает.

`--apropos -k` *ключевое_слово* Вывод заголовков страниц руководства, в которых содержится строка *ключевое_слово*. Список интересующих команд можно просмотреть. Этот ключ является эквивалентом команды apropos (см. раздел «apropos: Поиск по ключевому слову» на с. 55).

`--manpath=`*путь* `-M` *путь* Поиск man-страниц в каталогах, указанных в *пути*, где *путь* является списком каталогов, имена которых разделены двоеточиями. См. подраздел «Комментарии».

`--troff -t` Форматирование страницы для печати на PostScript-принтере. Информация поступает на стандартный вывод.

Комментарии

Страницы руководства сведены в разделы, каждый из которых посвящен отдельному аспекту системы Linux. В первом разделе содержатся команды, вызываемые пользователями, и этот раздел чаще всего посещается пользователями, не являющимися системными администраторами или программистами. Все остальные разделы руководства описывают системные вызовы, библиотечные функции и команды, используемые системными администраторами. Список разделов руководства представлен в разделе «man: Отображение системного руководства», в подразделе «Разделы руководства» на с. 54.

Разбиение на страницы Утилита man для отображения страниц, не помещающихся на один экран, использует утилиту less. Для использования другого средства

разбиения на страницы нужно установить в качестве значения переменной среды окружения PAGER полное имя этого средства. Например, добавление к содержимому файла ~/.bash_profile следующей строки позволит использоваться вместо less утилиту more:

```
export PAGER=/usr/bin/less
```

При работе под управлением OS X утилиты less и more являются копиями одного и того же файла. Но из-за способов вызова каждой из них в их работе имеются некоторые различия.

MANPATH Если переменной среды окружения MANPATH присвоить в качестве значения список каталогов с двоеточием в качестве разделителя, можно сообщить утилите man, где следует искать man-страницы. Например, пользователи оболочки bash, работающие в Linux, могут добавить к содержимому файла ~/.bash_profile следующую строку, которая заставит man вести поиск в каталогах /usr/man, /usr/local/man и /usr/local/share/man:

```
export MANPATH=/usr/man:/usr/local/man:/usr/local/share/man
```

При работе с правами привилегированного пользователя можно для дальнейшей настройки утилиты man отредактировать содержимое файла /etc/manpath.config или /etc/man.config (Linux) или файла /etc/man.conf (OS X). Дополнительные сведения даны на man-странице, посвященной самой утилите man.

Примечания

Утилита man также рассмотрена в разделе «man: Отображение системного руководства» на с. 53.

В качестве аргумента man может использоваться не только имя команды. Например, команда `man ascii` выведет список ASCII-символов и их различных представлений, а команда `man -k postscript` выведет список man-страниц, имеющих отношение к PostScript.

Обычно man-страницы хранятся в неотформатированной сжатой форме. При запросе man-страницы перед выводом на экран она должна быть восстановлена и отформатирована. Для ускорения обработки последующих запросов этой же man-страницы утилита man пытается сохранить отформатированную версию страницы.

Некоторые утилиты, описания которых даются на страницах руководства, имеют имена, совпадающие с именами встроенных команд оболочки. Поведение такой встроенной команды может слегка отличаться от поведения утилиты, описанного в man-странице. Сведения о встроенных командах оболочки даны на man-страницах, посвященных встроенным командам, или на man-странице, посвященной конкретной оболочке.

В ссылках на man-страницы часто используются номера разделов, стоящие в скобках. Например, `write(2)` ссылается на man-страницу для write в разделе 2 руководства (см. раздел «man: Отображение системного руководства» на с. 53).

В первой из следующих команд используется утилита col, генерирующая простую текстовую man-страницу, не содержащую жирного или подчеркнутого текста. Вторая команда генерирует PostScript-версию man-страницы.

```
$ man ls | col -b > ls.txt
$ man -t ls > ls.ps
```

При работе под управлением Linux для преобразования PostScript-файла в PDF-файл можно воспользоваться утилитой ps2pdf.

Примеры

В следующих примерах утилита man используется для вывода документации по команде write, которая отправляет сообщения на другой пользовательский терминал:

```
$ man write
WRITE(1)                  BSD General Commands Manual                  WRITE(1)
NAME
     write - send a message to another user
SYNOPSIS
     write user [ttyname]
DESCRIPTION
     The write utility allows you to communicate with other users, by copying
     lines from your terminal to theirs.
     When you run the write command, the user you are writing to gets a mes-
     sage of the form:
          Message from yourname@yourhost on yourtty at hh:mm ...
...
```

В следующем примере выводится man-страница для другой команды — самой команды man, которая может послужить неплохим начальным пособием для изучения системы:

```
$ man man
MAN(1)                        Manual pager utils                        MAN(1)
NAME
       man - an interface to the on-line reference manuals
SYNOPSIS
       man  [-c|-w|-tZ] [-H[browser]] [-T[device]] [-X[dpi]] [-adhu7V] [-i|-I]
       [-m system[,...]] [-L locale] [-p  string]  [-C  file]  [-M  path]  [-P
       pager]  [-r  prompt]  [-S  list] [-e extension] [--warnings [warnings]]
       [[section] page ...] ...
       ...

DESCRIPTION
       man  is  the  system's manual pager. Each page argument given to man is
       normally the name of a program, utility or function.  The  manual  page
       associated  with each of these arguments is then found and displayed. A
...
```

Утилиту man можно также использовать для поиска man-страниц, посвященных определенной теме. В следующем примере команда `man -k` выводит заголовки man-страниц, в которых содержится строка latex. Утилита apropos работает точно так же, как и команда `man -k`.

```
$ man -k latex
elatex (1) [latex]   - structured text formatting and typesetting
latex (1)            - structured text formatting and typesetting
mkindex (1)          - script to process LaTeX index and glossary files
pdflatex (1)         - PDF output from TeX
```

```
pod2latex (1)          - convert pod documentation to latex format
Pod::LaTeX (3perl)     - Convert Pod data to formatted Latex
...
```

Поиск, проводимый по ***ключевому_слову***, введенному вместе с ключом `-k`, не чувствителен к регистру букв. Хотя ***ключевое_слово***, введенное в командной строке, состоит только из букв нижнего регистра, оно соответствует последнему заголовку, содержащему строку `LaTeX` (с буквами в верхнем и нижнем регистрах). Запись `3perl` в последней строке показывает, что man-страница принадлежит разделу 3 — `Section 3 (Subroutines)` Системного руководства Linux и взята из справочного руководства для программистов, работающих на языке Perl — Perl Programmers Reference Guide (это подпрограмма на языке Perl, дополнительные сведения о языке программирования Perl даны в главе 11).

mkdir

Создание каталога.

*mkdir [ключ] **список_каталогов***

Утилита mkdir создает один или несколько каталогов.

Аргументы

Список_каталогов — это список путевых имен каталогов, создаваемых утилитой mkdir.

Ключи

При работе под управлением Linux утилита mkdir допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (`--`), работают только под управлением Linux. Однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--mode=`*режим* `-m` *режим* Установка прав доступа, указанных в *режиме*. Этот *режим* может быть представлен в абсолютной форме, использующей восьмеричное число (см. табл. V.7), или в символьной форме (см. табл. V.4).

`--parents -p` Создание каталогов, которые не существуют в пути к каталогу, который нужно создать.

`--verbose -v` Вывод имени каждого создаваемого каталога. Этот ключ применяется вместе с ключом `-p`.

Примечания

У пользователя, создающего каталог, должны быть права на запись и поиск (выполнение) в отношении родительского каталога, в котором создается новый каталог. Утилита mkdir создает каталоги, содержащие стандартные скрытые элементы (`.` и `..`).

Примеры

Следующая команда создает каталог accounts, который будет являться подкаталогом рабочего каталога, и каталог prospective в качестве подкаталога по отношению к каталогу accounts:

```
$ mkdir -p accounts/prospective
```

Теперь тот же самый пользователь, не меняя рабочего каталога, создает еще один подкаталог внутри каталога accounts:

```
$ mkdir accounts/existing
```

Затем пользователь меняет рабочий каталог на accounts и создает еще один подкаталог:

```
$ cd accounts
$ mkdir closed
```

В последнем примере показано, как пользователь создает еще один подкаталог. На этот раз ключ `--mode` удаляет все права доступа для группы и для всех остальных пользователей:

```
$ mkdir -m go= accounts/past_due
```

mkfs

Создание файловой системы на устройстве.

mkfs [ключи] устройство

Утилита mkfs создает файловую систему на таких устройствах, как гибкий диск или раздел жесткого диска. Она работает как внешний интерфейс для программ, которые создают файловые системы, каждая из которых приспособлена под тип своей файловой системы. Утилита mkfs доступна только при работе под Linux.

УТИЛИТА MKFS УНИЧТОЖАЕТ ВСЕ ДАННЫЕ НА УСТРОЙСТВЕ

При использовании утилиты mkfs будьте осторожны: она уничтожает все имеющиеся на устройстве данные.

Аргументы

Устройство — это имя устройства, на котором нужно создать файловую систему. Если имя устройства находится в каталоге /etc/fstab, то вместо самого имени можно использовать точку подключения устройства, то есть имя каталога, к которому оно примонтировано (например, /home вместо /dev/sda2).

Ключи

При запуске утилиты mkfs можно указать как глобальные ключи, так и ключи, специфические для той файловой системы, которую создает mkfs (например, для ext2, ext3, ext4, msdos, reiserfs). Глобальные ключи должны указываться раньше ключей, специфических для типа файловой системы.

Глобальные ключи

`-t` *fstype* (type — тип). Аргумент *fstype* является указанием типа той файловой системы, которую нужно создать, например ext2, ext3, msdos или reiserfs. По умолчанию могут использоваться указания на разные типы файловых систем.

`-v` (verbose — подробно). Вывод более подробной информации. Для специфической информации, касающейся конкретной файловой системы, следует использовать ключ `-V`.

Ключи, специфические для определенного типа файловой системы

Ключи, рассматриваемые в этом разделе, применяются для многих широко распространенных типов файловых систем, включая ext2 и ext3. Следующая команда выводит список утилит по созданию файловых систем, доступных на локальной системе:

```
$ ls /sbin/mkfs.*
/sbin/mkfs.bfs     /sbin/mkfs.ext2  /sbin/mkfs.minix  /sbin/mkfs.reiserfs
/sbin/mkfs.cramfs  /sbin/mkfs.ext3  /sbin/mkfs.msdos  /sbin/mkfs.vfat
```

Зачастую /sbin/mke2fs является ссылкой на /sbin/mkfs.ext2. Чтобы определить, какие специфические ключи применимы к тому или иному типу файловой системы, нужно просмотреть man-страницу или задать имя утилиты, создающей файловую систему.

```
$ /sbin/mkfs.ext3
Usage: mkfs.ext3 [-c|-l filename] [-b block-size] [-f fragment-size]
        [-i bytes-per-inode] [-I inode-size] [-J journal-options]
        [-N number-of-inodes] [-m reserved-blocks-percentage] [-o creator-os]
        [-g blocks-per-group] [-L volume-label] [-M last-mounted-directory]
        [-O feature[,...]] [-r fs-revision] [-E extended-option[,...]]
        [-T fs-type] [-jnqvFSV] device [blocks-count]
```

`-b` *размер* (block — блок). Указание размера блоков в байтах. В файловых системах ext2, ext3 и ext4 приемлемо использование блоков размером 1024, 2048 и 4096 байт.

`-c` (check — проверка). Проверки на сбойные блоки на устройстве перед созданием файловой системы. Для проведения медленного, разрушающего ранее имевшуюся информацию теста на чтение-запись этот ключ нужно указать дважды.

Комментарий

Перед тем как появится возможность записи на жесткий или гибкий диск или чтения с него данных в обычном порядке, на нем должна быть создана файловая система. Обычно жесткий диск разбит на разделы, каждый из которых имеет отдельную файловую систему. На гибком диске, как правило, содержится только одна файловая система. Более подробные сведения о файловых системах даны в главе 4.

Примечания

При работе под управлением Mac OS X для создания файловой системы используется утилита diskutil (см. соответствующее описание). Чтобы заменить существующую

файловую систему ext2 на журналируемую файловую систему типа ext3, нужно воспользоваться утилитой tune2fs (см. соответствующее описание) с ключом -j. (См. подраздел «Примеры».) Утилиту tune2fs можно также использовать для изменения частоты проверки файловой системы утилитой fsck (см. соответствующее описание).

mkfs является внешним интерфейсом

Подобно утилите fsck, утилита mkfs является внешним интерфейсом, вызывающим другие утилиты для работы с различными файловыми системами. Например, для создания получивших широкое распространение файловых систем ext2 и ext3 утилита mkfs вызывает утилиту mke2fs (которая обычно является ссылкой на утилиты mkfs.ext2 и mkfs.ext3). Дополнительные сведения даны на man-странице, посвященной утилите mke2fs. Другие утилиты, вызываемые mkfs, обычно называются mkfs.*mun*, где *mun* — это тип файловой системы. Благодаря подобному разбиению mkfs разработчики файловых систем могут предоставить программы для создания своих файловых систем, не оказывая при этом никакого влияния на разработчиков других файловых систем или не меняя способов использования mkfs со стороны системных администраторов.

Примеры

В следующем примере mkfs создает файловую систему на устройстве /dev/hda8. В данном случае по умолчанию используется тип файловой системы ext2.

```
# /sbin/mkfs /dev/sda5
mke2fs 1.41.4 (27-Jan-2009)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
122880 inodes, 489948 blocks
24497 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=67633152
60 block groups
8192 blocks per group, 8192 fragments per group
2048 inodes per group
Superblock backups stored on blocks:
        8193, 24577, 40961, 57345, 73729, 204801, 221185, 401409
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
This filesystem will be automatically checked every 37 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

Затем администратор воспользовался утилитой tune2fs для преобразования файловой системы ext2 в журналируемую файловую систему ext3:

```
# /sbin/tune2fs -j /dev/sda5
tune2fs 1.41.4 (27-Jan-2009)
Creating journal inode: done
This filesystem will be automatically checked every 37 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

Mtools

Использование команд в стиле DOS для работы с файлами и каталогами.

mcd [каталог]
mcopy [ключи] список_файлов цель
mdel список_файлов
mdir [–w] каталог
mformat [ключи] устройство
mtype [ключи] список_файлов

Рассматриваемые утилиты имитируют команды DOS и работают с файлами Linux, Mac OS X или DOS. Утилита mcopy предоставляет легкий способ перемещения файлов между файловой системой Linux/OS X и DOS-диском. По умолчанию устройством для всех команд служит /dev/fd0 или A:.

Утилиты

В табл. V.20 перечислены некоторые утилиты из коллекции Mtools.

Таблица V.20. Коллекция Mtools

Утилита	Функция
Mcd	Смена рабочего каталога на DOS-диске
Mcopy	Копирование DOS-файлов из одного каталога в другой
Mdel	Удаление DOS-файлов
Mdir	Вывод содержимого DOS-каталогов
Mformat	Добавление на диск информации о DOS-форматировании
Mtype	Вывод содержимого DOS-файлов

Аргументы

Каталогом, который используется с утилитами mcd и mdir, должно быть имя каталога на DOS-диске. ***Список_файлов***, который используется с утилитами mcopy и mtype, — это список, состоящий из имен файлов, разделенных пробелами. ***Цель***, которая используется с утилитой mcopy, — это имя обычного файла или каталога. Если передать утилите mcopy ***список_файлов***, содержащий более одного имени файла, то ***цель*** должна быть именем каталога. ***Устройство***, которое используется с утилитой mformat, — это буква DOS-привода, содержащего диск, который должен быть отформатирован (например, A:).

Ключи

mcopy

-n Автоматическая замена существующих файлов без выдачи запроса. Обычно mcopy перед перезаписью файла выдает запрос на подтверждение.

`-p` (preserve — сохранение). Сохранение атрибутов файлов при их копировании.

`-s` (recursive — рекурсивный режим). Копирование иерархий каталогов.

`-t` (text — текст). Преобразование текстовых файлов DOS для их использования на системах Linux/OS X, и наоборот. Строки в текстовых файлах DOS оканчиваются парами символов возврат каретки — перевод строки (разделитель строк) — RETURN–NEWLINE; строки в текстовых файлах Linux/OS X оканчиваются переводом строки (разделителем строк). Этот ключ удаляет символ перевода каретки при копировании из DOS-файла и добавляет его при копировании из Linux-файла.

mdir

`-w` (wide — широкий режим). Вывод только имен файлов, с помещением в каждую строку как можно больше этих имен. По умолчанию mdir выводит информацию о каждом файле на отдельной строке, показывая имя файла, размер и время создания файла.

mformat

`-f 1440` Указание 3,5-дюймовой дискеты высокой плотности (HD) емкостью 1440 Кб.

`-f 2880` Указание 3,5-дюймовой дискеты высокой плотности (HD) емкостью 2880 Кб.

`-v` *том* (label — метка). Присваивание метки тома *том* только что отформатированному DOS-диску.

mtype

`-t` (text — текст). Аналог ключа `-t` для утилиты mcopy. Этот ключ перед выводом файла заменяет каждую пару символов возврат каретки — перевод строки в DOS-файлах на один символ — разделитель строк.

Комментарий

Хотя эти утилиты имитируют поведение своих DOS-аналогов, они не претендуют на полное соответствие этим аналогам. В большинстве случаев те ограничения, которые накладываются DOS, снимаются. Например, нестрогая ссылка на имена файлов, представленная звездочкой (*), соответствует всем именам файлов (так же, как при работе в Linux/OS X), включая и те имена, для соответствия которым в DOS требуется применять набор групповых символов *.*.

Примечания

В данном описании термин DOS-диск относится либо к DOS-разделу жесткого диска, либо к гибкому диску DOS (дискете).

Если в хранилище дистрибутива Linux не окажется коллекции Mtools, эту коллекцию можно загрузить с главной страницы Mtools (www.gnu.org/software/mtools).

При работе под управлением Mac OS X для загрузки и компиляции Mtools нужно действовать поэтапно:

1. Установить MacPorts.
2. Выйти из всех запущенных терминалов.
3. Запустить терминал и запустить команду `sudo port install Mtools`.

Если локальное ядро настроено на поддержку файловых систем DOS, можно смонтировать DOS-диски на файловой системе Linux/OS X и работать с файлами, используя утилиты Linux/OS X. При всем удобстве этого свойства и исключении использования Mtools постоянное монтирование и размонтирование файловой системы DOS при каждой необходимости доступа к DOS-файлам вряд ли будет приемлемым с практической точки зрения. Подобные задачи могут стать слишком затратными по времени, к тому же некоторые системы настроены так, что рядовые пользователи не допускаются к монтированию и размонтированию файловых систем.

При использовании Mtools следует проявлять осмотрительность. Утилиты этой коллекции могут не предупреждать о том, что файл может быть переписан. Шанс переписать нужный файл можно понизить за счет использования вместо нестрогих ссылок на имена файлов конкретного указания путевых имен.

Чаще всего утилиты Mtools используются для проверки наличия файлов на дискетах DOS (mdir) и для копирования файлов между дискетой DOS и файловой системой Linux (mcopy). Идентифицировать DOS-диски можно путем использования обычных букв приводов, используемых в DOS: `A:` для первого привода гибких дисков, `C:` для первого жесткого диска и т. д. В качестве разделителя имен файлов в полном имени используется либо прямой слэш (/) как в Linux, либо обратный слэш (\) как в DOS. Чтобы не дать оболочке интерпретировать обратные слэши до передачи имени используемой утилите, их нужно отключить.

Каждая утилита из коллекции Mtools возвращает код выхода 0, если ее работа завершилась успешно, 1 — при полной и 2 — при частичной неудаче.

Примеры

В первом примере утилита mdir выводит содержимое дискеты DOS, находящейся в устройстве /dev/fd0:

```
$ mdir
 Volume in drive A is DOS UTY
 Directory for A:/
ACAD     LIF      419370    5-10-09   1:29p
CADVANCE LIF       40560    2-08-08  10:36a
CHIPTST  EXE        2209    4-26-09   4:22p
DISK     ID           31   12-27-09   4:49p
GENERIC  LIF       20983    2-08-08  10:37a
INSTALL  COM         896    7-05-09  10:23a
INSTALL  DAT       45277   12-27-09   4:49p
KDINSTAL EXE      110529    8-13-09  10:50a
LOTUS    LIF       44099    1-18-09   3:36p
PCAD     LIF       17846    5-01-09   3:46p
READID   EXE       17261    5-07-09   8:26a
README   TXT        9851    4-30-09  10:32a
```

```
UTILITY  LIF       51069    5-05-09   9:13a
WORD     LIF       16817    7-01-09   9:58a
WP       LIF       57992    8-29-09   4:22p
      15 File(s)     599040 bytes free
```

В следующем примере для копирования файлов *.TXT с дискеты DOS в рабочий каталог локальной файловой системы используется утилита mcopy. Поскольку расширение .TXT имеется только у одного файла, копируется только этот файл. Поскольку в DOS файлы с расширением .TXT обычно относятся к текстовым, ключ -t приводит к удалению ненужных символов возврата каретки (RETURN) в конце каждой строки.

Символ *, используемый в нестрогой ссылке на имена файлов, в командной строке отключается во избежание попытки со стороны оболочки расширить его еще до передачи аргумента утилите mcopy. Когда утилите mcopy передается шаблон *.txt, она находит файл README.TXT, поскольку DOS не делает различий между буквами верхнего и нижнего регистра, которые используются в именах файлов.

```
$ mcopy -t a:\*.txt .
Copying README.TXT
```

И в завершение дискета DOS переформатируется с помощью утилиты mformat, которая стирает все имеющиеся на ней данные. Если дискета не проходила низкоуровневое форматирование, то перед тем как ввести следующую команду, нужно воспользоваться утилитой fdformat:

```
$ mformat a:
```

Проверка с помощью утилиты mdir показывает, что дискета после форматирования абсолютно пуста:

```
$ mdir a:
Can't open /dev/fd0: No such device or address
Cannot initialize 'A:'
```

mv

Переименование или перемещение файла.

mv [ключи] существующий_файл новое_имя_файла
mv [ключи] список_существующих_файлов каталог
mv [ключи] существующий_каталог новый_каталог

Утилита mv, предназначенная для переименования или перемещения одного или нескольких файлов, имеет три формата. Первый формат используется для переименования одного файла с присвоением ему предоставленного вами нового имени файла. Второй формат используется для переименования одного или нескольких файлов с их появлением в указанном каталоге. Третий формат предназначен для переименования каталога. Если переименовать файл не удается, утилита mv осуществляет его физическое перемещение (то есть это происходит, если файл перемещается из одной файловой системы в другую).

Аргументы

В первой форме ***существующий_файл*** — это имя, указывающее на обычный файл, который нужно переименовать. ***Новое_имя_файла*** — это новое полное имя файла.

Во второй форме ***список_существующих_файлов*** — это список, состоящий из путевых имен файлов, которые нужно переименовать, а ***каталог*** указывает на новый родительский каталог для файлов. Файлы, которые подвергаются переименованию, будут иметь такие же простые имена, как и каждый файл в ***списке_существующих_файлов***, но у них будут новые абсолютные имена.

Третья форма используется для переименования ***существующего_каталога*** и присвоения ему имени ***новый_каталог***. Эта форма работает только в том случае, если каталога ***новый_каталог*** еще не существует.

Ключи

При работе под управлением Linux утилита mv допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме специально оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--backup -b` Создание резервной копии (путем добавления к имени файла символа ~) любого файла, который будет переписан. (Linux)

`--force -f` Принуждение mv к отказу от выдачи предупреждения, если при перемещении будет переписан существующий файл, права на запись в который у вас отсутствуют. У вас должны быть права на запись в каталог, в котором содержится существующий файл.

`--interactive -i` Вывод приглашения на подтверждение перезаписи файла при работе утилиты mv. Если ответ на это приглашение будет начинаться с «y» или «Y», mv перепишет файл, в противном случае mv не станет перемещать файл.

`--update -u` Если при перемещении будет переписываться существующий файл, а не каталог, этот ключ заставит mv сравнить время последнего изменения исходного и целевого файлов. Если у целевого файла время последнего изменения свежее (целевой файл новее исходного), mv не станет его заменять. (Linux)

`--verbose -v` Вывод списка перемещаемых файлов.

Примечания

Когда GNU-утилита mv копирует файл из одной файловой системы в другую, она действует как утилита cp (с ключом `-a`) и утилита rm: сначала ***существующий_файл*** копируется в файл с именем ***новое_имя_файла***, а затем ***существующий_файл*** удаляется. Если файл с именем ***новое_имя_файла*** уже существует, то перед копированием mv может его удалить.

Как и в случае вызова утилиты rm, у вас должны быть права на запись и выполнение в отношении родительского каталога ***существующего_файла***, но права на чтение самого файла или на запись в него не нужны. Если перемещение должно привести к перезаписи файла, право записи в который отсутствует, mv выводит права доступа к файлу и ждет ответа. Если ввести «y» или «Y», mv переписывает файл, в противном случае файл не перемещается. Если используется ключ `-f`, mv не выводит приглашение, требующее ответа, а просто переписывает файл.

Хотя ранние версии mv могли перемещать между файловыми системами только обычные файлы, теперь mv может перемещать файлы любых типов, включая каталоги и файлы устройств.

Примеры

Первая команда изменяет имя файла letter, который находится в рабочем каталоге, на letter.1201:

```
$ mv letter letter.1201
```

Следующая команда изменяет имя файла таким образом, что он с тем же простым именем оказывается в пользовательском каталоге ~/archives:

```
$ mv letter.1201 ~/archives
```

Следующая команда перемещает все файлы в рабочем каталоге, чьи имена начинаются с memo, в каталог /p04/backup:

```
$ mv memo* /p04/backup
```

Использование ключа -u не дает утилите mv заменять более новый файл более старым. После выполнения показанной ниже команды mv -u более новый файл (memo2) остается не переписанным. Команда mv без ключа -u переписывает более новый файл (время последнего изменения файла memo2 и его размер изменяются на такие же, которые были у файла memo1).

```
$ ls -l
-rw-rw-r--  1 sam sam 22 Mar 25 23:34 memo1
-rw-rw-r--  1 sam sam 19 Mar 25 23:40 memo2
$ mv -u memo1 memo2
$ ls -l
-rw-rw-r--  1 sam sam 22 Mar 25 23:34 memo1
-rw-rw-r--  1 sam sam 19 Mar 25 23:40 memo2
$ mv memo1 memo2
$ ls -l
-rw-rw-r--  1 sam sam 22 Mar 25 23:34 memo2
```

nice

Изменение уровня приоритета команды.

nice [ключ] [командная_строка]

Утилита nice сообщает о приоритете оболочки или изменяет приоритет команды. Обычный пользователь может понизить приоритет команды. Повысить ее приоритет может только пользователь, работающий с root-правами. Синтаксис встроенной в оболочку TC Shell команды nice отличается от синтаксиста этой утилиты. Дополнительные сведения даны в подразделе «Примечания».

Аргументы

Командная_строка — это командная строка, которую нужно выполнить с другим уровнем приоритета. Без ключей и аргументов nice выводит уровень приоритета оболочки, запустившей nice.

Ключи

Без ключей nice использует настройку по умолчанию, равную 10, которая понижает уровень приоритета команды до 10: обычно выбирается значение от 0 до 10. С увеличением этого значения приоритет, с которым запускается команда, падает.

Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--adjustment=`*значение*
`-n` *значение*
: Смена приоритета путем инкремента (или декремента) на указанное *значение*. Приоритет имеет диапазон от –20 (самый высокий) до 19 (самый низкий). Положительное *значение* понижает приоритет, а отрицательное *значение* его повышает. Отрицательное *значение* может указывать только пользователь, работающий с root-правами. При указании *значения* вне этого диапазона приоритет устанавливается по его границе.

Примечания

Для изменения уровня приоритета запущенного процесса можно воспользоваться утилитой renice (см. ее описание) или командой `r` утилиты top (см. описание этой утилиты).

Более высокие (более положительные) значения приоритета означают, что ядро планирует выполнение задания менее часто. Более низкие (более отрицательные) значения означают, что выполнение задания планируется чаще.

Когда пользователь, работающий с root-правами, планирует задание для работы с самым высоким приоритетом, это может повлиять на производительность системы в отношении всех остальных заданий, включая саму операционную систему. Поэтому применять утилиту nice с отрицательными значениями нужно весьма осмотрительно.

В оболочке TC Shell есть встроенная команда nice. При работе под управлением оболочки tcsh для изменения приоритета, с которым запускается ***командная_строка***, нужно использовать следующий синтаксис. По умолчанию уровень приоритета равен 4. Для положительных значений нужно включать знак плюс.

nice [±значение] командная_строка

Встроенная в tcsh команда nice работает не так, как утилита nice: при использовании встроенной команды `nice -5` уменьшает приоритет, с которым запускается ***командная_строка***. А при использовании утилиты `nice -n -5` увеличивает приоритет, с которым запускается ***командная_строка***.

Примеры

Следующая команда выполняет поиск с помощью утилиты find в фоновом режиме с наименьшим возможным приоритетом. Команда `ps -l` выводит nice-значение этой команды в столбце `NI`:

```
# nice -n 19 find / -name core -print > corefiles.out &
[1] 422
# ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 R     0   389  8657  0  80   0 -  4408 -      pts/4    00:00:00 bash
```

```
4 D     0   422   389 28  99  19 -  1009 -      pts/4    00:00:04 find
0 R     0   433   389  0  80   0 -  1591 -      pts/4    00:00:00 ps
```

Следующая команда занимается поиском очень больших файлов и запускается с высоким приоритетом (–15):

```
# nice -n -15 find / -size +50000k
```

nohup

Запуск команды, которая продолжает свою работу после вашего выхода из системы.

nohup командная_строка

Утилита nohup выполняет командную строку таким образом, что команда продолжает выполняться после вашего выхода из системы. Иными словами, nohup заставляет процесс игнорировать сигнал SIGHUP. В зависимости от настройки локальной оболочки она может прекращать работу фоновых процессов. В оболочке TC Shell имеется встроенная команда nohup. Дополнительные сведения даны в подразделе «Примечания».

Аргументы

Командная_строка — это та самая командная строка, которую нужно выполнить.

Примечания

При работе под управлением Linux утилита nohup допускает применение общих ключей, описание которых дано в табл. V.2.

Если не перенаправить вывод из той команды, которая запускается с помощью утилиты nohup, то и стандартный вывод, и стандартная ошибка отправляются в файл по имени nohup.out, находящийся в рабочем каталоге. При отсутствии прав записи в рабочий каталог nohup отправляет вывод в файл ~/nohup.out.

В отличие от утилиты nohup, встроенная в оболочку TC Shell команда nohup не отправляет вывод в файл nohup.out. Фоновые задания, запущенные из tcsh, продолжают выполняться и после вашего выхода из системы.

Примеры

Следующая команда выполняет поиск в фоновом режиме, используя утилиту nohup:

```
$ nohup find / -name core -print > corefiles.out &
[1] 14235
```

od

Вывод содержимого файла.

od [ключи] [список_файлов]

Утилита od (octal dump — восьмеричный дамп) выводит содержимое файла. Дамп нужен для просмотра исполняемых (объектных) файлов и текстовых файлов со встав-

ками из невыводимых на печать символов. Эта утилита получает ввод из файла, указанного в командной строке, или из стандартного ввода. Утилита od доступна только при работе под управлением Linux.

Аргументы

Список_файлов указывает путевые имена тех файлов, содержимое которых отображается с помощью утилиты od. Если ***список_файлов*** не указан, od читает данные из стандартного ввода.

Ключи

Утилита od допускает применение общих ключей, описание которых дано в табл. V.2.

`--address-radix=`*база*
`-A` *база* — Указывает *базу*, используемую при выводе смещений, показываемых для позиций в файле. По умолчанию смещения даются в виде восьмеричных чисел. Возможными значениями для *базы* являются `d` (decimal — десятичная), `o` (octal — восьмеричная), `x` (hexadecimal — шестнадцатеричная) и `n` (без вывода смещений).

`--skip-bytes=`*n*
`-j` *n* — Пропуск *n* байтов перед выводом данных.

`--read-bytes=`*n*
`-N` *n* — Чтение максимально возможного количества из *n* байт с последующим завершением работы.

`--strings=`*n*
`-S` *n* — Вывод из файла только тех байтов, которые содержат последовательность из *n* или более выводимых на печать ASCII-символов, в конце которых стоит нулевой байт (NULL).

`--format=`*тип*`[`*n*`]`
`-t` *тип* `[`*n*`]` — Указание формата вывода данных из файла. Для просмотра содержимого файла в нескольких разных форматах этот ключ можно применять повторно, указывая типы этих форматов. Возможные значения для *типа* перечислены в табл. V.21.

По умолчанию утилита od выводит дамп файла в виде двухбайтовых восьмеричных чисел. Количество байт, используемое od для составления каждого числа, можно указать, задав индикатор длины *n*. Этот индикатор можно задать для всех типов, за исключением `a` и `c`. Возможные значения *n* перечислены в табл. V.23.

Таблица V.21. Форматы вывода

Тип	Тип вывода
`a`	Именованный символ. Вывод невыводимых на печать управляющих символов путем использования их официальных ASCII-названий. Например, подача страницы (FORMFEED) выводится как `ff`
`c`	ASCII-символ. Вывод невыводимых на печать управляющих символов в виде эскейп-последовательностей, использующих обратный слэш (см. табл. V.22) или трехразрядных восьмеричных чисел
`d`	Десятичное число со знаком
`f`	Число с плавающей точкой

продолжение

Таблица V.21 (*продолжение*)

Тип	Тип вывода
o	Восьмеричное число (используется по умолчанию)
u	Беззнаковое десятичное число
x	Шестнадцатеричное число

Таблица V.22. Эскейп-последовательности, использующие обратный слэш для типа формата вывода «c»

Последовательность	Значение
\0	Нуль (NULL)
\a	Звонок (BELL)
\b	Забой (BACKSPACE)
\f	Подача страницы (FORMFEED)
\n	Разделитель строк (NEWLINE)
\r	Возврат каретки (RETURN)
\t	Горизонтальная табуляция (HORIZONTAL TAB)
\v	Вертикальная табуляция (VERTICAL TAB)

Таблица V.23. Индикаторы длины

n	Используемое количество байт
Целые числа (типы d, o, u и x)	
C (character — символ)	Использует по одному символу для каждого десятичного значения
S (short integer — короткое целое число)	Использует 2 байта
I (integer — целое число)	Использует 4 байта
L (long — длинное число)	Использует 4 байта на 32-разрядных машинах и 8 байт на 64-разрядных машинах
Числа с плавающей точкой (тип f)	
F (float —число с плавающей точкой)	Использует 4 байта
D (double — с двойной точностью)	Использует 8 байт
L (long double — длинное с двойной точностью)	Как правило, использует 8 байт

Примечания

Для поддержки обратной совместимости с устаревшими версиями od, не придерживающихся стандартов POSIX, в утилите od используются ключи, перечисленные в табл. V.24, являющиеся сокращенными версиями многих ранее рассмотренных ключей.

Таблица V.24. Сокращенные спецификации формата

Сокращение	Эквивалентная спецификация
-a	-t a
-b	-t oC

Сокращение	Эквивалентная спецификация
-c	-t c
-d	-t u2
-f	-t fF
-h	-t x2
-i	-t d2
-l	-t d4
-o	-t o2
-x	-t x2

Примеры

Файл ac, используемый в следующих примерах, содержит все ASCII-символы. В первом примере байты в этом файле отображаются как именованные символы.

В первом столбце показывается смещение первого байта каждой строки вывода от начала файла. Смещения выводятся в виде восьмеричных значений.

```
$ od -t a ac
0000000 nul soh stx etx eot enq ack bel bs ht nl vt ff cr so si
0000020 dle dc1 dc2 dc3 dc4 nak syn etb can em sub esc fs gs rs us
0000040 sp ! " # $ % & ' ( ) * + , - . /
0000060 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
0000100 @ A B C D E F G H I J K L M N O
0000120 P Q R S T U V W X Y Z [ \ ] ^ _
0000140 ` a b c d e f g h i j k l m n o
0000160 p q r s t u v w x y z { | } ~ del
0000200 nul soh stx etx eot enq ack bel bs ht nl vt ff cr so si
0000220 dle dc1 dc2 dc3 dc4 nak syn etb can em sub esc fs gs rs us
0000240 sp ! " # $ % & ' ( ) * + , - . /
0000260 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
0000300 @ A B C D E F G H I J K L M N O
0000320 P Q R S T U V W X Y Z [ \ ] ^ _
0000340 ` a b c d e f g h i j k l m n o
0000360 p q r s t u v w x y z { | } ~ del
0000400 nl
0000401
```

В следующем примере байты выводятся как восьмеричные числа, ASCII-символы или печатные символы, перед которыми ставится обратный слэш (см. табл. V.22):

```
$ od -t c ac
0000000 \0 001 002 003 004 005 006 \a \b \t \n \v \f \r 016 017
0000020 020 021 022 023 024 025 026 027 030 031 032 033 034 035 036 037
0000040 ! " # $ % & ' ( ) * + , - . /
0000060 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
0000100 @ A B C D E F G H I J K L M N O
0000120 P Q R S T U V W X Y Z [ \ ] ^ _
0000140 ` a b c d e f g h i j k l m n o
0000160 p q r s t u v w x y z { | } ~ 177
0000200 200 201 202 203 204 205 206 207 210 211 212 213 214 215 216 217
0000220 220 221 222 223 224 225 226 227 230 231 232 233 234 235 236 237
```

```
0000240 240 241 242 243 244 245 246 247 250 251 252 253 254 255 256 257
0000260 260 261 262 263 264 265 266 267 270 271 272 273 274 275 276 277
0000300 300 301 302 303 304 305 306 307 310 311 312 313 314 315 316 317
0000320 320 321 322 323 324 325 326 327 330 331 332 333 334 335 336 337
0000340 340 341 342 343 344 345 346 347 350 351 352 353 354 355 356 357
0000360 360 361 362 363 364 365 366 367 370 371 372 373 374 375 376 377
0000400 \n
0000401
```

В последнем примере в файле /usr/bin/who ищутся все строки, имеющие длину не менее трех символов (норма, используемая по умолчанию) и заканчивающиеся нулевым байтом. Еще один способ вывода такого же списка показан в описании утилиты strings. Позиции строк показаны в виде десятичных, а не восьмеричных смещений.

```
$ od -A d -S 3 /usr/bin/who
…
0028455 /usr/share/locale
0028473 Michael Stone
0028487 David MacKenzie
0028503 Joseph Arceneaux
0028520 6.10
0028525 GNU coreutils
0028539 who
0028543 abdlmpqrstuwHT
0028558 %Y-%m-%d %H:%M
0028573 %b %e %H:%M
0028585 extra operand %s
0028602 all
0028606 count
0028612 dead
0028617 heading
0028625 ips
0028629 login
…
```

open (OS X)

Открытие файлов, каталогов и URL-ресурсов.

open [ключ] [список_файлов]

Утилита open открывает один или несколько файлов, каталогов или URL-ресурсов.

Аргументы

Список_файлов указывает путевые имена файлов каталогов или URL-ресурсов, открываемых с помощью утилиты open.

Ключи

Без ключей open открывает файлы в ***списке_файлов*** как по двойному щелчку на значке каждого файла в Finder.

`-a` *приложение* Открытие ***списка_файлов*** с использование приложения. Этот ключ является эквивалентом перетаскивания ***списка_файлов*** на значок приложения в Finder.

`-b` *пакет* Открытие ***списка_файлов*** с использованием приложения с идентификатором пакета ***пакет***. Идентификатор пакета — это строка, зарегистрированная системой, идентифицирующая приложение, которое может открывать файлы. Например, идентификатор пакета com.apple.TextEdit указывает на редактор TextEdit.

`-e` (edit — редактировать). Открытие ***списка_файлов*** с использованием TextEdit.

`-f` (file — файл). Открытие стандартного ввода как файла в текстовом редакторе, используемом по умолчанию. Этот ключ не допускает применения ***списка_файлов***.

`-t` (text — текст). Открытие ***списка_файлов*** в текстовом редакторе, используемом по умолчанию (см. подраздел «Комментарий»).

Комментарий

При открытии файла используется приложение, связанное с этим файлом. Например, открытие файла образа диска приводит к его монтированию. Команда open сразу же возвращает управление, не дожидаясь запуска приложения.

LaunchServices — это системная среда, занимающаяся идентификацией приложений, которые могут открывать файлы. Она обрабатывает список доступных приложений и учитывает предпочтения пользователя относительно того, какое приложение нужно использовать для каждого типа файла. LaunchServices также отслеживает, какой текстовый редактор нужно использовать при вызове утилиты с ключами `-t` и `-f`.

Примечания

Когда файл будет открыт с помощью приложения, работающего с использованием графического пользовательского интерфейса (GUI), утилиту open нужно запускать из эмулятора Terminal или другого эмулятора терминала, запускаемого под управлением GUI. В противном случае операция потерпит неудачу.

Примеры

В первом примере монтируется файл образа диска backups.dmg. Диск монтируется к каталогу /Volumes с использованием имени, под которым он был отформатирован.

```
$ ls /Volumes
House   Spare   Leopard
$ open backups.dmg
$ ls /Volumes
Backups   House     Spare     Leopard
```

В следующей команде открывается файл picture.jpg. Этот и следующий пример нужно запускать из текстового окна в графическом интерфейсе (например, в Terminal). Выбираемое приложение зависит от атрибутов файла. Если тип файла и код создателя

указывают на конкретное приложение, утилита open открывает файл, используя это приложение. В противном случае open использует для обработки файлов .jpg имеющуюся в системе программу по умолчанию.

```
$ open picture.jpg
```

В последнем примере в Finder открывается каталог /usr/bin. Каталог /usr в Finder обычно не показывается, потому что имеет установленный флаг атрибута невидимости. Но команда open может открыть любой файл, к которому имеется доступ из оболочки, даже если он обычно недоступен из Finder.

```
$ open /usr/bin
```

otool (OS X)

Вывод объектных, библиотечных и выполняемых файлов.

otool ***ключи список_файлов***

Утилита otool выводит информацию об объектном, библиотечном или исполняемом файле или отображает часть этого файла.

Аргументы

Список_файлов указывает путевые имена файлов, выводимых с помощью otool.

Ключи

Для указания той части каждого файла из ***списка_файлов***, которая будет выведена утилитой otool, нужно использовать хотя бы один из ключей `-L`, `-M`, `-t` или `-T`.

`-L` (libraries — библиотеки). Вывод имен и номеров версий общих библиотек, используемых объектным файлом.

`-M` (module — модуль). Вывод таблица модулей общей библиотеки.

`-p` *имя* (print — печать). Начало вывода с символа по имени *имя*. Этот ключ требует применения ключа `-t` и одного из ключей `-v` или `-V`.

`-T` (table of contents — список содержимого). Вывод списка содержимого общей библиотеки.

`-t` (text — текст). Вывод секции `TEXT` объектного файла.

`-V` (very verbose — очень подробно). Более подробный (чем с ключом `-v`) вывод данных в символьном виде. При выводе кода этот ключ заставляет otool выводить имена вызываемых подпрограмм, а не их адреса.

`-v` (verbose — подробно). Вывод данных в символьном виде. При выводе кода этот ключ заставляет otool выводить имена инструкций, а не числовые коды.

Комментарий

Утилита otool выводит информацию о содержимом и о зависимостях объектных файлов. Эта информация может принести пользу при отладке программы. Например, при

настройке файла chroot утилита otool может сообщить, какие библиотеки нужны для запуска данной программы.

Некоторые ключи применяются только с конкретными типами объектных модулей. Например, для обычного исполняемого файла использование ключа `-T` не приведет к выводу какой-либо информации.

Примечания

Утилита otool является частью дополнительно устанавливаемого инструментария разработчика Developer Tools.

На системах, использующих двоичный формат ELF, команда `otool -L` делает то же самое, что и утилита ldd.

Примеры

Примеры в этом разделе используют скомпилированную версию следующей программы на языке C:

```
$ cat myname.c
#include <stdio.h>
int main(void) {
    printf("My name is Sam.\n");
    return 0;
}
```

В первом примере otool выводит имена библиотек, с которыми связана программа:

```
$ otool -L myname
myname:
   /usr/lib/libmx.A.dylib (compatibility version 1.0.0, current version 92.0.0)
   /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 88.0.0)
```

В некоторых случаях библиотека, используемая программой, сама зависит от других библиотек. Чтобы посмотреть, не использует ли она другие библиотеки, можно использовать команду `otool -L`:

```
$ otool -L /usr/lib/libmx.A.dylib /usr/lib/libSystem.B.dylib
/usr/lib/libmx.A.dylib:
   /usr/lib/libmx.A.dylib (compatibility version 1.0.0, current version 92.0.0)
   /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 88.0.0)
/usr/lib/libSystem.B.dylib:
   /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 88.0.0)
   /usr/lib/system/libmathCommon.A.dylib (compatibility version 1.0.0, current …
```

В следующем примере дизассемблируется код функции `main`. При компиляции программы компилятор иногда изменяет символьные имена. Компилятор дает таким функциям, как `main`, первый знак подчеркивания, после чего получается символьное имя `_main`.

```
$ otool -Vt -p _main myname
myname:
(__TEXT,__text) section
_main:
00002ac0        mfspr   r0,lr
00002ac4        stmw    r30,0xfff8(r1)
```

```
00002ac8        stw     r0,0x8(r1)
00002acc        stwu    r1,0xffb0(r1)
00002ad0        or      r30,r1,r1
00002ad4        bcl     20,31,0x2ad8
00002ad8        mfspr   r31,lr
00002adc        addis   r2,r31,0x0
00002ae0        addi    r3,r2,0x4b8
00002ae4        bl      _printf$LDBLStub
00002ae8        li      r0,0x0
00002aec        or      r3,r0,r0
00002af0        lwz     r1,0x0(r1)
00002af4        lwz     r0,0x8(r1)
00002af8        mtspr   lr,r0
00002afc        lmw     r30,0xfff8(r1)
00002b00        blr
...
```

paste

Объединение соответствующих строк из файлов.

paste [ключ] [список_файлов]

Утилита paste читает строки из файлов, указанных в ***списке_файлов***, и объединяет соответствующие строки в своем выводе. По умолчанию выводимые строки разделяются символом табуляции.

Аргументы

Список_файлов — это список имен обычных файлов. Если ***список_файлов*** опустить, paste осуществляет чтение данных из стандартного ввода.

Ключи

При работе под управлением Linux утилита paste допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--delimiter=`*dlist* `-d` *dlist* — Аргумент *dlist* — это список символов, используемых в качестве разделителей полей вывода. Если *dlist* содержит один символ, paste использует этот символ для разделения полей вместо используемого по умолчанию символа табуляции. Если в *dlist* содержится более одного символа, символы для разделения полей используются по очереди, а затем, если это необходимо, заново используются с начала списка.

`--serial -s` — Обработка файлов по одному с горизонтальной вставкой данных. См. подраздел «Примеры».

Примечания

Утилита paste часто используется для перестановки столбцов таблицы. Такая утилита, как cut, может помещать нужные столбцы в отдельные файлы, а затем утилита paste может объединять их в любом другом порядке.

Примеры

В следующем примере используются файлы по имени fnames и acctinfo. Эти файлы можно создать, используя утилиту cut (см. соответствующее описание) и файл /etc/passwd. Команда paste ставит на первое место поле полного имени — fullname, за которым следует вся остальная информация об учетной записи пользователя. Два поля вывода разделяет символ табуляции. Хотя этот пример работает и под управлением Mac OS X, файл /etc/passwd не содержит информации о большинстве пользователей.

```
$ cat fnames
Sam the Great
Max Wild
Zach Brill
Helen Simpson

$ cat acctinfo
sam:x:401:50:/home/sam:/bin/zsh
max:x:402:50:/home/max:/bin/bash
zach:x:504:500:/home/zach:/bin/tcsh
hls:x:505:500:/home/hls:/bin/bash

$ paste fnames acctinfo
Sam the Great   sam:x:401:50:/home/sam:/bin/zsh
Max Wild        max:x:402:50:/home/max:/bin/bash
Zach Brill      zach:x:504:500:/home/zach:/bin/tcsh
Helen Simpson   hls:x:505:500:/home/hls:/bin/bash
```

В следующих примерах используются файлы p1, p2, p3 и p4. В последнем примере в этой группе с помощью ключа –d утилите paste передается список символов для использования их в качестве разделителей полей вывода:

```
$ cat p1
1
one
ONE
$ cat p2
2
two
TWO
extra
$ cat p3
3
three
THREE
$ cat p4
4
```

```
four
FOUR

$ paste p4 p3 p2 p1
4       3       2       1
four    three   two     one
FOUR    THREE   TWO     ONE
                extra

$ paste -d="+-=" p3 p2 p1 p4
3+2-1=4
three+two-one=four
THREE+TWO-ONE=FOUR
+extra-=
```

В последнем примере используется ключ `--serial`, который приводит к поочередной вставке файлов:

```
$ paste --serial p1 p2 p3 p4
1       one     ONE
2       two     TWO     extra
3       three   THREE
4       four    FOUR
```

pax

Создание архива, восстановление файлов из архива или копирование иерархии каталога.

pax [ключи] [список_шаблонов]
pax –w [ключи] [файлы-источники]
pax –r [ключи] [список_шаблонов]
pax –rw [ключи] [файлы-источники] каталог-приемник

Утилита pax имеет четыре режима работы: режим вывода списка демонстрирует список файлов в архиве, режим создания помещает несколько файлов в один архивный файл, режим извлечения восстанавливает файлы из архива и режим копирования копирует иерархию каталога. Архивные файлы, созданные с помощью утилиты pax, могут быть сохранены на диске, сменном носителе или на удаленной системе.

Главный ключ определяет режим, в которой запускается pax. Режим вывода списка (задаваемый отсутствием главного ключа) заключается в чтении архива со стандартного ввода и выводе списка файлов, хранящихся в архиве. Режим создания (`-w` для записи) заключается в чтении списка, состоящего из имен обычных файлов или имен файлов каталогов из командной строки или из стандартного ввода, и в записи создаваемого архивного файла на стандартный вывод. Режим извлечения (`-r` для чтения) заключается в чтении архива со стандартного ввода и в извлечении файлов из этого архива. Можно восстановить из архива все файлы или только те из них, чьи имена соответствуют шаблону. Режим копирования (`-rw`) заключается в чтении списка, состоящего из имен обычных файлов или имен файлов каталогов из командной строки или со стандартного ввода и копировании файлов в существующий каталог.

Аргументы

В режиме создания утилита pax читает имена файлов, которые должны быть включены в архив из ***файлов-источников*** в командной строке. Если ***файлы-источники*** отсутствуют, pax читает имена со стандартного ввода, по одному имени в строке.

По умолчанию в режиме извлечения pax извлекает все файлы из архива, считанного со стандартного ввода, а в режиме вывода списка показывает имена этих файлов. Можно задать выборочное извлечение файлов или выборочный вывод их имен, предоставив команде ***список_шаблонов***. Если имя файла в архиве соответствует шаблону из ***списка_шаблонов***, pax извлекает этот файл или выводит имя этого файла, в противном случае файл игнорируется. Шаблоны, используемые при вызове утилиты pax, аналогичны групповым символам оболочки (см. раздел «Генерация имени файла и расширение имени» на с. 157), за исключением того, что они соответствуют в именах файлов слэшам (/) и начальным точкам (.). Дополнительные сведения о шаблонах pax даны в man-странице, посвященной утилите fnmatch (только для OS X).

В режиме копирования утилита не создает ***каталог-приемник***. Этот каталог должен существовать до вызова pax в режиме копирования.

Ключи

Главный ключ задает режим работы утилиты pax: создания, извлечения или копирования.

Главные ключи

Режим работы утилиты pax определяется тремя ключами. В командную строку нужно включить один из них или вообще не указывать главный ключ. Без главного ключа pax работает в режиме вывода списка.

`-r` (read — чтение). Чтение архива со стандартного ввода и извлечение файлов. Если ***список_шаблонов*** в командной строке не указан, утилита pax извлекает из архива все файлы. При использовании ***списка_шаблонов*** pax извлекает только те файлы, чьи имена соответствуют одному из шаблонов в ***списке_шаблонов***. В следующем примере из архивного файла root.pax, находящегося на внешнем носителе, извлекаются только те файлы, чьи имена оканчиваются на .c:

```
$ pax -r \*.c < /Volumes/Backups/root.pax
```

Обратный слэш не дает оболочке расширить символ * до передачи аргумента утилите pax.

`-w` (write — запись). Создание архива из файлов, имена которых указываются в командной строке или поступают со стандартного ввода. Эти файлы могут быть как обычными файлами, так и файлами каталогов. Если имена файлов поступают со стандартного ввода, каждое из них должно быть на отдельной строке. Архив записывается в стандартный вывод. Имена файлов, используемые pax, часто генерируются утилитой find. Следующая команда создает архив из каталога /Users и записывает его в архивный файл по имени Users.pax в каталоге /Volumes/Backups:

```
# find -d /Users -print | pax -w > /Volumes/Backups/Users.pax
```

Ключ -d заставляет утилиту find искать файлы, начиная с тех, которые расположены глубже всех, уменьшая вероятность возникновения проблем с правами доступа при восстановлении файлов из архива. Этот ключ утилиты find рассмотрен в ее описании.

-rw (copy — копирование). Копирование файлов, имена которых указаны в командной строке или поступают со стандартного ввода из одного места системы в другое. Вместо создания архивного файла, содержащего файлы, чьи имена заданы со стандартного ввода, pax копирует эти файлы в ***каталог-приемник*** (указанный в качестве последнего аргумента командной строки pax). Получаемый эффект аналогичен созданию архива в режиме создания с последующем извлечении из него файлов в режиме извлечения, за исключением того, что режим копирования позволяет обойтись без создания архива.

В следующем примере иерархия рабочего каталога копируется в каталог /Users/max/code. Каталог code должен существовать до ввода этой команды. Нужно убедиться в том, что ~max/code не является подкаталогом рабочего каталога, иначе будет создана рекурсивная копия.

```
$ pax -rw . ~max/code
```

Другие ключи

Следующие ключи изменяют поведение pax. Эти ключи работают с одним или несколькими главными ключами.

-c (complement — дополнение). Инвертирование смысла теста, осуществляемого с помощью ***списка_шаблонов***. Перечисляются или помещаются в архив только те файлы, чьи имена не соответствуют ни одному из шаблонов в ***списке_шаблонов***.

-f *архив* (file — файл). Использование *архива* как имени архивного файла. В режимах вывода списка и извлечения этот ключ заставляет читать данные из *архива*, а не со стандартного ввода. В режиме создания он заставляет вести запись в *архив*, а не в стандартный вывод. Этот ключ можно использовать для доступа по сети к устройству на другой системе. Дополнительные сведения даны в описании ключа --file для утилиты tar (см. соответствующий раздел).

-H (частичное разыменование). Для каждого файла, являющегося символьной ссылкой, копирование файла, на который указывает ссылка, а не о самой символьной ссылки. Действие этого ключа распространяется на файлы, указанные в командной строке, оно не касается файлов, найденных при спуске по иерархии каталога. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример использования этого ключа приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L».

-L (разыменование). Для каждого файла, являющегося символьной ссылкой, вывод копирование файла, на который указывает ссылка, а не о самой символьной ссылке. Действие этого ключа распространяется на все файлы. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример использования этого ключа приведен в разделе «chgrp», в подразделе «Сравнение ключей –H и –L»

`-l` (link — ссылка). В режиме копирования создание, по возможности, жестких ссылок вместо копирования файлов.

`-P` (без разыменования). Для каждого файла, являющегося символьной ссылкой, копирование символьной ссылки, а не файла, на который она указывает. Действие этого ключа распространяется на все файлы. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Такое поведение является для pax действием по умолчанию. Пример использования этого ключа приведен в разделе «chgrp», в подразделе «Примеры».

`-p` *список_сохраняемых* Сохраняет или сбрасывает атрибуты файла, указанные в *списке_сохраняемых*, который является строкой из одной или нескольких букв, показанный в табл. V.25. По умолчанию pax сохраняет информацию о времени последнего обращения к файлу и времени внесения в него последних изменений, но не сохраняет сведения о владельце или о правах доступа к файлу. Этот ключ работает только в режимах извлечения и копирования.

Таблица V.25. Сохранение флагов

Буква	Значение
a	Сброс времени последнего обращения
e	Сохранение всех атрибутов файла
m	Сброс времени последнего внесения изменений в файл
o	Сохранение сведений о владельце
p	Сохранение прав доступа

`-s` *подкоманда* Выполнение *подкоманды* в отношении имен файлов при работе с ними в любом из режимов. *Подкоманда* имеет следующий формат:

*s/**строка_поиска**/строка_замены**/[gp]*

Утилита pax заменяет соответствия регулярному выражению *строка_поиска* *строкой_замены*. Замыкающая буква g задает глобальную замену, без нее pax заменяет в каждом имени файла только первую строку, соответствующую *строке_поиска*. Замыкающая буква p (означающая print — печать) заставляет pax вывести каждую выполняемую замену. *Подкоманда* работает аналогично свойству поиска и замены редактора vim, рассмотренному в разделе «Поиск и замена строк» на с. 197, за исключением того, что ей не нужен адрес.

`-v` (verbose — подробно). В режиме вывода списка вывод информации, аналогичной той, которая выводится командой `ls -l`. В других режимах вывод списка обрабатываемых файлов.

`-X` В режимах копирования и создания запрещение утилите pax вести просмотр каталогов в файловой системе, за исключением тех, в которых содержатся ***файлы-источники***.

`-x` *формат* В режиме создания запись архива в *формате*, показанном в табл.V.26. Если формат не указан, pax записывает файл в tar-формате POSIX (в таблице этот *формат* показан как ustar).

Таблица V.26. Форматы архива

формат	**Описание**
cpio	Формат, указываемый в POSIX для cpio-архивов
sv4cpio	Формат, используемый для cpio-архивов при работе под управлением UNIX System V, выпуск 4
tar	Исторический Berkeley tar-формат
ustar	POSIX tar-формат (используется по умолчанию)

-z (gzip). В режиме создания сжатие архивов с использованием утилиты gzip. В режиме извлечения восстановление файлов архива с помощью утилиты gunzip.

Комментарий

Утилита pax является основной заменой для tar, cpio и других программ архивирования. В режиме создания и копирования pax обрабатывает указанные каталоги в рекурсивном режиме. В режимах вывода списка и извлечения, если шаблон в ***списке_шаблонов*** соответствует имени каталога, pax выводит список или извлекает файлы из этого каталога.

Примечания

У pax нет своего собственного формата. Утилита pax может читать и записывать архивы в нескольких форматах. По умолчанию она создает архивы в POSIX tar-формате. Список поддерживаемых pax форматов показан в табл. V.26.

Утилита pax сама определяет формат архивного файла, не разрешая указывать формат архива (ключ -x) в режимах вывода списка или извлечения.

При работе под управлением Mac OS X версии 10.4 и более поздних версий pax копирует расширенные атрибуты.

Примеры

В первом примере pax создает архив по имени corres.0901.pax. В этом архиве хранится содержимое каталога corres.

```
$ pax -w corres > corres.0901.pax
```

Ключ –w переводит pax в режим создания, где список файлов, помещаемых в архив, предоставляется аргументами командной строки (в предыдущем примере – corres) или получается со стандартного ввода. Утилита pax отправляет архив на стандартный вывод. В предыдущем примере вывод перенаправляется в файл по имени corres.0901.pax.

Далее, без использования ключей pax выводит список файлов, имеющихся в архиве, который она читает со стандартного ввода:

```
$ pax < corres.0901.pax
corres
corres/hls
corres/max
corres/max/0823
```

```
corres/max/0828
corres/max/0901
corres/memo1
corres/notes
```

Следующая команда использует ключ -f для задания имени файла ввода и выполнения той же функции, что и в предыдущей команде:

```
$ pax -f corres.0901.pax
```

Когда pax как в предыдущем примере читает архивный, она определяет формат файла самостоятельно. Указывать этот формат не нужно (и не разрешено).

В следующем примере, работающем под управлением Mac OS X, предпринимается попытка создания архива иерархии каталога /etc в используемом по умолчанию tar-формате. В данном случае утилита pax запущена пользователем, имеющим root-права, поскольку некоторые файлы в этой иерархии не могут быть прочитаны обычными пользователями. Поскольку pax не следует по умолчанию по символьным ссылкам (не занимается разыменованием) и поскольку при работе под управлением Mac OS X /etc является символьной ссылкой, pax копирует ссылку, а не каталог, на который она указывает. Первая команда создает архив, вторая команда показывает ссылку в архиве, а третья команда использует ключ -v, чтобы показать, что это ссылка:

```
# pax -w -f /tmp/etc.tar /etc
# pax -f /tmp/etc.tar
/etc
# pax -v -f /tmp/etc.tar
lrwxr-xr-x  1 root     admin          0 May 21 01:48 /etc => private/etc
pax: ustar vol 1, 1 files, 10240 bytes read, 0 bytes written.
```

Ключ -L заставляет следовать по ссылкам (заниматься разыменованием). В следующем примере pax создает требуемый архив из содержимого каталога, на который указывает /etc:

```
# pax -wLf /tmp/etc.tar /etc
# pax -f /tmp/etc.tar
/etc
/etc/6to4.conf
/etc/AFP.conf
/etc/afpovertcp.cfg
/etc/aliases
/etc/aliases.db
/etc/amavisd.conf
/etc/amavisd.conf.personal
/etc/appletalk.cfg
...
```

В следующем примере pax используется для создания резервной копии каталога memos, сохраняя сведения о владельце и правах доступа к файлу. ***Каталог-приемник*** должен существовать еще до того, как утилита pax будет запущена в режиме копирования.

```
$ mkdir memos.0625
$ pax -rw -p e memos memos.0625
$ ls memos.0625
memos
```

В предыдущем примере каталог memos копируется в ***каталог-приемник***. Утилита pax может использоваться для копирования каталога в рабочий каталог без помещения его в подкаталог. В следующем примере ключ –s заставляет pax заменить имя каталога memos именем . (именем рабочего каталога) в процессе записи файлов в каталог memos.0625:

```
$ pax -rw -p e -s /memos/./ memos memos.0625
```

В последнем примере для создания списка файлов, имена которых начинаются со строки memo и которые находятся в иерархии рабочего каталога, используется утилита find. Утилита pax записывает эти файлы в архив в cpio-формате. Вывод из утилиты pax направляется на стандартный вывод, который переправляется через канал утилите bzip2, которая сжимает его перед записью в архивный файл.

```
$ find . -type f -name "memo*" | pax -w -x cpio | bzip2 > memos.cpio.bz2
```

plutil (OS X)

Работа с файлами списков свойств.

plutil [ключи] список_файлов

Утилита plutil переводит файлы списка свойств из одного формата в другой и проверяет их синтаксис.

Аргументы

Список_файлов используется для указания путевых имен одного или нескольких файлов, с которыми работает утилита plutil.

Ключи

`-convert` *формат* — Конвертирование файлов из ***списка_файлов*** в указанный *формат*, который может быть либо `xml1`, либо `binary1`.

`-e` *расширение* — Присваивание файлам вывода имен с указанным *расширением*.

`-help` — Вывод справки.

`-lint` — Проверка синтаксиса файлов (используется по умолчанию).

`-o` *файл* — (output — вывод). Присваивание конвертированному файлу имени *файл*.

`-s` — (silent — молчание). Отказ от вывода любых сообщений для успешно конвертированных файлов.

Комментарий

Утилита plutil конвертирует файлы из XML-формата списка свойств в двоичный формат, и наоборот. Утилита plutil может читать, но не записывать устаревший простой текстовый формат списка свойств.

Примечания

Утилита plutil допускает применение двойного дефиса (--) в качестве метки конца ключей в командной строке. Дополнительная информация дана в разделе «Общие ключи».

Примеры

В следующем примере проверяется синтаксис файла по имени java.plist:

```
$ plutil java.plist
java.plist: OK
```

В следующем примере показывается вывод plutil при проверке с ее помощью поврежденного файла со списком свойств:

```
$ plutil broken.plist
broken.plist:
XML parser error:       Ошибка XML-парсера:
    Encountered unexpected element at line 2 (plist can only include one object)
    В строке 2 найден непредусмотренный элемент (plist может включать только один
    объект)
Old-style plist parser error:   Ошибка парсера, работающего с устаревшим форматом:
    Malformed data byte group at line 1; invalid hex
    В строке 1 содержится деформированная группа байтов данных;
    неверное шестнадцатеричное число
```

В следующем примере файл StartupParameters.plist конвертируется в двоичный формат, при этом исходный файл переписывается:

```
$ plutil -convert binary1 StartupParameters.plist
```

В завершающем примере файл списка свойств loginwindow.plist, имеющий двоичный формат и находящийся в рабочем каталоге, конвертируется в XML-формат и благодаря использованию ключа -o конвертированный файл помещается в файл по имени /tmp/lw.p:

```
$ plutil -convert xml1 -o /tmp/lw.p loginwindow.plist
```

pr

Разбиение файлов на страницы для последующего вывода на печать.

pr [ключи] [список_файлов]

Утилита pr разбивает файлы на страницы, как правило, перед их выводом на печать. У каждой страницы имеется заголовок с именем файла, датой, временем и номером страницы.

Утилита pr получает ввод из файлов, указанный в командной строке или со стандартного ввода. Вывод из pr отправляется на стандартный вывод, и он зачастую перенаправляется через канал на принтер.

Аргументы

Список_файлов — это список путевых имен текстовых файлов, разбиваемых с помощью pr на страницы. Если опустить *список_файлов*, pr читает данные со стандартного ввода.

Ключи

При работе под управлением Linux утилита pr допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

Ключи можно вставить в ***список_файлов***. Действие вставленных ключей распространяется только на те файлы, имена которых следуют за ними в командной строке.

`--show-control-chars` `-c` — Вывод управляющих символов со знаком вставки (^; например, ^H). Другие не выводимые на печать символы выводятся в виде восьмеричных чисел, перед которыми ставится обратный слэш. (Linux)

`--columns=`***col*** `-`***col*** — Вывод данных в *col* столбцах, по умолчанию в одном столбце. Этот ключ может привести к усечению строк и не может использоваться вместе с ключом `-m` (`--merge`).

`--double-space` `-d` — Вывод с двойными интервалами.

`--form-feed` `-F` — Использование символа перевода страницы (FORMFEED) для пропуска листа до следующей страницы вместо заполнения текущей страницы символами разделителя строк (NEWLINE).

`--header=`***заголовок*** `-h` ***заголовок*** — Вывод *заголовка* в верхней части каждой страницы вместо имени файла. Если *заголовок* содержит пробелы, его нужно взять в кавычки.

`--length=`***строки*** `-l` ***строки*** — Установка длины страницы в *строках*. По умолчанию устанавливается длина в 66 строк.

`--merge -m` — Вывод всех указанных файлов одновременно в нескольких столбцах. Этот ключ не может быть использован с ключом `-col` (`--columns`).

`--number-lines=[`***c[num]]*** `-n`***[c[num]]*** — Нумерация строк вывода. Утилита pr добавляет символ *c* к номеру, чтобы отделить его от содержимого файла (по умолчанию для этого используется символ табуляции). Аргумент *num* определяет количество цифр в каждом номере строки (по умолчанию оно равно 5).

`--indent=`***пробелы*** `-o` ***пробелы*** — Создание отступа за счет пробелов (указание левой границы).

`--separator=c` `-s[c]` — Разделение столбцов одиночным символом *c* (по умолчанию, если опустить *c*, используется символ табуляции). По умолчанию *pr* использует символы табуляции в качестве символов-разделителей для выравнивания столбцов, пока не используется ключ `-w`, при котором столбцы вообще ничем не разделяются.

`--omit-header` `-t` — Принуждение pr к тому, чтобы не выводить пятистрочный заголовок страницы и ее окончание. Заголовок, который обычно выводит pr, включает имя файла, дату, время и номер страницы. Окончание состоит из пяти пустых строк.

`--width=`*num*
`-w` *num*

Установка ширины страницы в *num* столбцов. Этот ключ работает только с выводом в несколько столбцов (когда используется ключ `-m` (`--merge`) или ключ `-col` (`--columns`)).

`--pages=`*первая_страница*`[:`*последняя_страница*`]`
`+` *первая_страница*`[:`*последняя_страница*`]`

Вывод начинается со страницы с номером *первая_страница* и заканчивается страницей с номером *последняя_страница*. Без указания *последней_страницы* утилита pr выводит все до последней страницы документа. Короткая версия этого ключа начинается не с дефиса, а со знака плюс. Mac OS X не воспринимает аргумент *последняя_страница*; печать всегда продолжается до конца документа.

Примечания

При использовании ключа `-col` (`--columns`) для вывода данных в нескольких столбцах pr выводит одинаковые номера строк в каждом столбце (с возможным исключением последнего).

Примеры

Первая команда показывает утилиту pr, которая разбивает на страницы файл по имени memo и отправляет свой вывод через канал на устройство lpr для его распечатки:

```
$ pr memo | lpr
```

Затем memo опять отправляется на принтер, на этот раз со специальным заголовком в верхней части каждой страницы. Задание запускается в фоновом режиме.

```
$ pr -h 'MEMO RE: BOOK' memo | lpr &
[1] 4904
```

И в завершение pr выводит файл memo на экран без заголовка, начиная со страницы номер 3:

```
$ pr -t +3 memo
...
```

ps

Вывод состояния процесса.

ps [ключи] [список_процессов]

Утилита ps выводит информацию о состоянии процессов, запущенных на локальной системе.

Аргументы

Список_процессов — это разделенный пробелами список PID-номеров. Если использовать ***список_процессов***, ps выводит отчет только о тех процессах, которые указаны в списке.

Ключи

При работе под управлением Linux утилита ps допускает использование трех типов ключей, перед каждым из которых ставится разный префикс. Ключи можно перемешивать. Дополнительные сведения даны на man-странице, посвященной ps.

Два дефиса: GNU (длинные) ключи.

Один дефис: UNIX98 (короткие) ключи.

Без дефисов: BSD ключи.

Ключи, перед которыми ставится двойной дефис (--) или перед которым дефис вообще не ставится, работают только под Linux. Однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X

`-A` (all — все). Вывод отчета обо всех процессах. Аналог ключа -e.

`-e` (everything — все). Вывод отчета обо всех процессах. Аналог ключа -A.

`-f` (full — полный). Вывод листинга, содержащего больше столбцов с информацией.

`--forest f` Вывод дерева процессов (перед f дефис отсутствует). (Linux)

`-l` (long — длинный). Вывод длинного листинга, показывающего больше информации о каждом процессе. Описание столбцов, отображаемых с помощью этого ключа, дано в подразделе «Комментарий».

`--no-headers` Пропуск заголовка. Этот ключ применяется в том случае, если нужно отправить вывод другой программе. (Linux)

`--user` *имена_пользователей - имена_пользователей* Вывод отчета о процессах, запущенных пользователями, чьи имена или UID-номера перечислены в аргументе *имена_пользователей* с использованием двоеточия в качестве разделителя. Может быть указано одно или несколько имен пользователей, вошедших на локальную систему.

`-w` (wide — широкий). Без этого ключа ps усекает строки с правой стороны экрана. Этот ключ приводит к расширению отображения до 132 позиций, которые заворачиваются, если нужно, еще на одну строку. Для расширения отображения на неограниченное количество позиций этот ключ используется дважды; именно этот режим используется по умолчанию, если вывод утилиты ls подвергается перенаправлению.

Комментарий

Без ключей ps выводит состояния всех активных процессов, управляемых вашим терминалом или экраном. В табл. V.27 перечислены заголовки и содержимое каждого из четырех столбцов, выводимых утилитой ps.

Таблица V.27. Заголовки столбцов, часть I

Заголовок	Значение
PID	Идентификационный номер процесса
TTY (terminal)	Название терминала, управляющего процессом

Заголовок	Значение
TIME	Количество часов, минут и секунд, показывающее время запуска процесса
CMD	Командная строка, с которой был вызван процесс. Команда усекается, чтобы поместиться на одной строке. Чтобы увидеть остальную часть командной строки, нужно использовать ключ –w

Столбцы, выводимые утилитой ps, зависят от выбранных вами ключей. В табл. V.28 перечислены заголовки и содержимое наиболее часто выводимых столбцов.

Таблица V.28. Заголовки столбцов, часть II. (Заголовки столбцов различаются в зависимости от типа примененного ключа. В этой таблице показаны заголовки для ключей UNIX98 (с одним дефисом).)

Заголовок	Значение
%CPU	Процент, используемый процессом от всего времени центрального процессора. Из-за способа вычислений, проводимых в Linux/OS X в отношении процессов, этот показатель дается приблизительно, а сумма значений %CPU может превышать 100%
%MEM (память)	Процент используемой процессом оперативной памяти (RAM)
COMMAND *или* CMD	Командная строка, с которой был вызван процесс. Команда усекается чтобы поместиться на одной строке. Чтобы увидеть остальную часть командной строки, нужно использовать ключ –w. Этот столбец всегда показывается в конце строки
F (флаги)	Флаги, связанные с процессом
NI (значение nice)	Значение nice процесса (см. описание утилиты nice)
PID	Идентификационный номер процесса
PPID (родительский PID)	Идентификационный номер родительского процесс
PRI (приоритет)	Приоритет процесса
RSS (resident set size — размер резидентного набора)	Количество блоков памяти, используемое процессом
SIZE или SZ	Размер в блоках образа оперативной памяти процесса
STIME или START	Дата запуска процесса
STAT или S (состояние)	Состояние процесса, указанное одной или несколькими буквами из следующего списка: < Высокоприоритетный D Приостановлен и не поддается прерыванию L Страницы заблокированы в памяти (в режиме реального времени и пользовательского ввода-вывода) N Низкоприоритетный R Доступный для выполнения (в очереди на запуск) S Приостановлен T Либо остановленный, либо трассируемый W Не имеющий резидентных страниц в оперативной памяти X Неработающий Z Зомби-процесс, ожидающий, пока его дочерний процесс будет завершен

продолжение

Таблица V.28 (*продолжение*)

Заголовок	Значение
TIME	Количество минут и секунд со времени запуска процесса
TTY (терминал)	Название терминала, управляющего процессом
USER или UID	Имя пользователя, владеющего процессом
WCHAN (wait channel — ожидание канала)	Если процесс ожидает наступление события, адрес функции ядра, заставляющей процесс находиться в режиме ожидания. Для процессов, не находящихся в режиме ожидания или приостановки, это значение равно нулю

Примечание

Для динамического отображения информации о состоянии процесса следует использовать утилиту top (см. соответствующее описание).

Примеры

В первом примере показано использование утилиты ps без ключей, выводящей сведения об активных процессах пользователя. Первый процесс — это работа оболочки (bash), а второй — выполнение утилиты ps.

```
$ ps
  PID TTY          TIME CMD
 2697 pts/0    00:00:02 bash
 3299 pts/0    00:00:00 ps
```

С ключом -l (long — длинный) ps выводит более полную информацию о процессах:

```
$ ps -l
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
000 S   500  2697  2696  0  75   0    -   639 wait4  pts/0    00:00:02 bash
000 R   500  3300  2697  0  76   0    -   744 -      pts/0    00:00:00 ps
```

Ключ -u заставляет показывать информацию, касающуюся указанного пользователя:

```
$ ps -u root
  PID TTY          TIME CMD
    1 ?        00:00:01 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 migration/0
    4 ?        00:00:01 ksoftirqd/0
    5 ?        00:00:00 watchdog/0
...
```

Ключ --forest заставляет ps выводить информацию, которая в man-странице описывается как «ASCII art process tree», то есть «Дерево процессов, нарисованное с помощью ASCII-символов». Процессы, являющиеся дочерними по отношению к другим процессам, появляются с отступом ниже своих родительских процессов, создавая иерархию процессов или дерево, которое легче просматривается.

```
$ ps -ef --forest
UID        PID  PPID  C STIME TTY          TIME CMD
```

```
root         1     0  0 Jul22 ?        00:00:03 init
root         2     1  0 Jul22 ?        00:00:00 [keventd]
…
root       785     1  0 Jul22 ?        00:00:00 /usr/sbin/apmd -p 10 -w 5 -W -P
root       839     1  0 Jul22 ?        00:00:01 /usr/sbin/sshd
root      3305   839  0 Aug01 ?        00:00:00  \_ /usr/sbin/sshd
max       3307  3305  0 Aug01 ?        00:00:00      \_ /usr/sbin/sshd
max       3308  3307  0 Aug01 pts/1    00:00:00          \_ -bash
max       3774  3308  0 Aug01 pts/1    00:00:00              \_ ps -ef --forest
…
root      1040     1  0 Jul22 ?        00:00:00 login -- root
root      3351  1040  0 Aug01 tty2     00:00:00  \_ -bash
root      3402  3351  0 Aug01 tty2     00:00:00      \_ make modules
root      3416  3402  0 Aug01 tty2     00:00:00          \_ make -C drivers CFLA
root      3764  3416  0 Aug01 tty2     00:00:00              \_ make -C scsi mod
root      3773  3764  0 Aug01 tty2     00:00:00                  \_ ld -m elf_i3
```

ps и kill Следующая последовательность команд показывает, как использовать ps для определения PID-номеров процессов, запущенных в фоновом режиме, и как определить, какой процесс использует команду kill. В данном случае не обязательно использовать ps, поскольку оболочка выводит PID-номер фонового процесса. Утилита ps подтверждает PID-номер.

Первая команда выполняет find в фоновом режиме. Оболочка показывает задание и PID-номер процесса, за которыми следует приглашение на ввод команды.

```
$ find ~ -name memo -print > memo.out &
[1] 3343
```

Затем ps подтверждает PID-номер фоновой задачи. Если этот номер был неизвестен, то единственным средством его получения служит команда ps.

```
$ ps
  PID TTY          TIME CMD
 3308 pts/1    00:00:00 bash
 3343 pts/1    00:00:00 find
 3344 pts/1    00:00:00 ps
```

И наконец, команда kill (см. соответствующее описание) завершает процесс:

```
$ kill 3343
$ RETURN
[1]+  Terminated              find ~ -name memo -print >memo.out
$
```

rcp

Копирование одного или нескольких файлов на удаленную систему или из нее.

rcp [ключи] файл-источник файл-приемник
rcp [ключи] список_ файлов_источников каталог-приемник

Утилита rcp копирует один или несколько обычных файлов между двумя системами, которые могут осуществлять обмен данными по сети.

RCP НЕ ОБЕСПЕЧИВАЕТ ДОЛЖНУЮ СТЕПЕНЬ БЕЗОПАСНОСТИ

При работе под управлением Linux утилита mkdir допускает применение общих ключей, описание которых дано в табл. V.2.

Аргументы

Файл-источник, ***список_файлов_источников*** и ***файл-приемник*** — это путевые имена обычных файлов, а ***каталог-приеменик*** — это имя файла каталога. Полное имя, не содержащее двоеточия (:), — это имя файла на локальной системе. Имя, составленное по форме *имя@хост:путь*, является именем файла на удаленной системе по имени *host*. Элемент *path* указывается относительно личного каталога пользователя, указанного в элементе *имя* (если только *путь* не является абсолютным путевым именем). Если часть приемника *имя@* опущена, относительное имя привязывается к имеющемуся на хосте личному каталогу пользователя, вызвавшего команду rcp.

Как и утилита cp, rcp имеет два режима работы: в первом один файл копируется в другой, а во втором один или несколько файлов копируются в каталог. ***Файл-источник*** или ***список_ файлов_источников*** — это имя или список имен файлов, которые будут скопированы утилитой rcp, ***файл-приемник*** — это имя, которое rcp присвоит получившейся копии файла, а ***каталог-приемник*** — это имя каталога, в который rcp поместит копируемые файлы. Если rcp копирует файлы в ***каталог-приемник***, эти файлы сохраняют свои прежние простые имена.

Ключи

`-p` (preserve — сохранение). Установка для каждой копии таких же показателей времени последнего изменения и информации о правах доступа, которые были у ***файла-источника***. Если ключ `-p` не используется, то rcp на удаленной системе использует для изменения прав доступа к файлу маску, применяемую ко вновь создаваемым файлам (см. описание утилиты umask).

`-r` (recursive — рекурсивный режим). Когда файл в ***списке_файлов_источников*** является каталогом, копирование содержимого этого каталога и всех подкаталогов в ***каталог-приемник***. Этот ключ можно использовать, только если в качестве места назначения используется каталог.

Примечания

Чтобы копировать файлы на удаленную систему или из нее с помощью утилиты rcp, на ней нужно иметь учетную запись. Утилита rcp не выводит приглашение на ввод пароля, используя вместо этого несколько альтернативных методов для проверки наличия прав на чтение или запись файлов на удаленной системе.

Один из методов требует, чтобы имя локальной системы было указано в файле /etc/hosts.equiv на удаленной системе. Если это имя там присутствует, rcp разрешает пользователю копировать файлы, если его пользовательское имя одинаково на обеих системах, и учетная запись на удаленной системе предполагает наличие необходимых прав доступа к файлам на ней.

Предоставление прав доступа может быть также указано индивидуально для каждого пользователя. При использовании данного метода личный каталог удаленного

пользователя должен содержать файл по имени ~/.rhosts, в котором перечислены доверенные удаленные системы и пользователи. Имя пользователя на локальной и удаленной системах не должны быть одинаковым, но имя на локальной системе, которое начинается с имени самой локальной системы, должно быть указано в строке в удаленном файле ~/.rhosts. Образец файла .rhosts приведен в описании утилиты rlogin.

При использовании в имени на удаленной системе групповых символов (например, *) нужно цитировать групповой символ или имя, чтобы групповой символ не был интерпретирован оболочкой на удаленной системе, а также оболочкой на локальной системе. Как и в случае использования утилиты cp, если ***файл-приемник*** существует до выполнения rcp, эта утилита переписывает файл.

Примеры

В первом примере файлы с именами, оканчивающимися на .c, копируются в каталог archives, который находится на удаленной системе по имени bravo. Поскольку в команде не указывается имя пользователя, rcp использует имя локального пользователя, имеющееся на удаленной системе. Поскольку в команде не указано полное имя каталога archives, утилита rcp предполагает, что это подкаталог личного каталога пользователя на системе bravo. Каждый копируемый файл сохраняет свое простое имя.

```
$ rcp *.c bravo:archives
```

В следующем примере файл memo из каталога /home/sam на системе bravo копируется в рабочий каталог на локальной системе:

```
$ rcp bravo:/home/sam/memo .
```

Затем утилита rcp копирует файлы по имени memo.new и letter в личный каталог пользователя Sam на удаленной системе bravo. Абсолютными путевыми именами копируемых файлов на bravo являются соответственно /home/sam/memo.new и /home/sam/letter:

```
$ rcp memo.new letter bravo:/home/sam
```

И последняя команда копирует все файлы в каталоге report на системе bravo, который принадлежит пользователю Sam, в каталог oldreports на локальной системе, сохраняя при этом в данных копиях исходные показатели времени внесения последних изменений и информацию о правах доступа:

```
$ rcp -p 'bravo:reports/*' oldreports
```

renice

Изменение приоритета процесса

renice ***приоритет список_процессов***
renice ***–n инкремент список_процессов***

Утилита renice изменяет приоритет запущенных процессов. Обычный пользователь может уменьшить приоритет процесса, запущенного им самим. Увеличить приоритет процесса или изменить приоритет процесса, запущенного другим пользователем, может только пользователь, работающий с root-правами.

Аргументы

В ***списке_процессов*** указываются PID-номера процессов, приоритет которых должен быть изменен. Для каждого процесса устанавливается указанный ***приоритет***, или, если используется второй формат, его приоритет получает приращение, соответствующее значению ***инкремента*** (которое может быть и отрицательным).

Ключи

Ключи, которые могут быть указаны в любом месте ***списка_процессов***, изменяют толкование аргументов, которые следуют за ними в командной строке.

`-p` (process — процесс). Интерпретация следующих аргументов в качестве идентификационных номеров процессов (PID) (используется по умолчанию).

`-u` (user — пользователь). Интерпретация следующих аргументов в качестве имен пользователей или пользовательских ID-номеров.

Примечания

Диапазон приоритетов простирается от –20 (самый высокий) до +20 (самый низкий). Более высокие (более положительные) значения приоритета означают, что ядро распределяет заданию процессорное время менее часто. Более низкие (более отрицательные) значения заставляют распределять заданию процессорное время более часто.

Когда пользователь работает с root-правами, планирует выполнение задания с наивысшим приоритетом, это изменение может повлиять на производительность системы в отношении всех остальных заданий, включая саму операционную систему. Поэтому при использовании утилиты renice с отрицательными значениями следует проявлять особую осмотрительность.

Если нужно запустить процесс с приоритетом, отличным от присваиваемого по умолчанию, следует воспользоваться утилитой nice (см. соответствующее описание).

Примеры

В первом примере уменьшается приоритет всех задач, владельцем которых является пользователь Zach:

```
$ renice -n 5 -u zach
```

В следующем примере пользователь с root-правами использует утилиту ps для проверки приоритета процесса, в котором запущена утилита find. Столбец NI (nice) показывает значение 19, и администратор решает повысить приоритет на 5 единиц:

```
# ps -l
UID   PID  PPID CPU PRI NI      VSZ    RSS WCHAN  STAT  TT       TIME COMMAND
501  9705  9701   0  31  0    27792    856 -      Ss    p1    0:00.15 -bash
501 10548  9705   0  12 19    27252    516 -      RN    p1    0:00.62 find /
# renice -n -5 10548
```

rlogin

Вход на удаленную систему.

rlogin [ключ] удаленная_система

Утилита rlogin создает рабочий сеанс на удаленной системе по сети.

RLOGIN НЕ ОБПЕСПЕЧИВАЕТ ДОЛЖНУЮ СТЕПЕНЬ БЕЗОПАСНОСТИ

Для авторизации вашего входа в систему утилита rlogin использует доверие на уровне хостов, которое не обеспечивает должную степень безопасности. Время от времени она отправляет ваш пароль по сети в виде простого текста, что противоречит нормам безопасности. Лучше по возможности пользоваться утилитой ssh (см. соответствующее описание).

Аргументы

Удаленная_система — это имя той системы, к которой локальная система может получить доступ по сети.

Ключи

`-l`*имя_пользователя* (login). Регистрация на удаленной системе как пользователя с указанным *именем_пользователя*.

Примечания

Если в находящемся на удаленной системе файле по имени /etc/hosts.equiv указано имя локальной системы, удаленная система не станет выводить приглашение на ввод пароля. Системы, перечисленные в файле /etc/hosts.equiv, считаются такими же безопасными, как и локальная система.

Другой способ указания доверительных отношений базируется на индивидуальной для каждого пользователя основе. В личном каталоге каждого пользователя может содержаться файл по имени ~/.rhosts, в котором имеется список доверенных удаленных систем и пользователей. Образец файла .rhosts приведен в подразделе «Примеры».

Примеры

Следующий пример иллюстрирует использование утилиты rlogin. На локальной системе пользователь Max использует имя пользователя max; на удаленной системе bravo его именем пользователя является watson. Удаленная система приглашает пользователя Max ввести пароль, поскольку он входит в систему, используя имя пользователя, отличающееся от того, которое он использует на локальной системе.

```
$ who am i
max      tty06        Oct 14 13:26
$ rlogin -l watson bravo
Password:
```

Файл ~/.rhosts Если локальная система называется hurrah, то следующий файл .rhosts на системе bravo позволяет пользователю max войти в систему под именем watson без ввода пароля:

```
$ cat ~watson/.rhosts
hurrah max
```

rm

Удаление файла (удаление ссылки).

rm [ключи] список_файлов

Утилита rm удаляет жесткие и (или) символьные ссылки на один или несколько файлов. Если для файла удаляется последняя жесткая ссылка, удаляется и сам файл.

ПРИ ИСПОЛЬЗОВАНИИ RM С ГРУППОВЫМИ СИМВОЛАМИ СЛЕДУЕТ ПРОЯВЛЯТЬ ОСОБУЮ ОСМОТРИТЕЛЬНОСТЬ

Поскольку эта утилита позволяет удалять большое количество файлов одной командой, пользоваться rm нужно очень осмотрительно, особенно когда работа ведется с нестрогими ссылками на имена файлов. Если есть хоть малейшие сомнения по поводу результатов работы команды rm с нестрогой ссылкой на имена файлов, сначала нужно воспользоваться командой echo с той же ссылкой на имена файлов и получить список файлов, генерируемых ссылкой. Можно также воспользоваться ключом –i (– –interactive) для работы в интерактивном режиме.

Аргументы

Список_файлов — это список тех файлов, которые будут удалены с помощью утилиты rm.

Ключи

При работе под управлением Linux утилита rm допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--force -f` Удаление файлов, для которых у вас нет прав на запись, без запроса вашего согласия. Этот ключ также приводит к подавлению информационных сообщений, если файл не существует.

`--interactive -i` Запрос перед удалением каждого файла. Если с этим ключом используется ключ `--recursive`, rm также выдает запрос перед проверкой каждого каталога.

`--recursive -r` Удаление содержимого указанного каталога, включая все его подкаталоги и сам каталог. Этот ключ следует использовать с особой осмотрительностью.

`--verbose -v` Вывод имени каждого удаляемого файла.

Примечания

Чтобы удалить файл, нужны права на выполнение и чтение его родительского каталога, но не нужны права на чтение или запись в отношении самого файла. Если утилита rm запущена в интерактивном режиме (то есть если стандартный ввод на rm поступает с клавиатуры) и отсутствуют право записи в файл, rm выводит права доступа и ждет

ответа. Если ответ начинается с «y» или «Y», rm удаляет файл; в противном случае она бездействует. Если стандартный ввод поступает не с клавиатуры, rm удаляет файл без всякого запроса.

Информация о жестких ссылках дана в главе 4, в разделе «Жесткие ссылки», а информация о символьных ссылках дана в разделе «Символьные ссылки» на с. 130. Вопрос удаления ссылок рассмотрен в разделе «rm: Удаление ссылки» на с. 132. Если нужно удалить пустой каталог, используется утилита rmdir (см. соответствующее описание).

Когда нужно удалить файл, имя которого начинается с дефиса, нужно заставить rm не рассматривать имя файла в качестве ключа. Одним из способов является использование перед именем файла специального ключа -- (двойной дефис). Этот ключ сообщает rm, что за ним уже нет никаких ключей: все остальные аргументы являются именами файлов, даже если они выглядят как ключи.

ДЛЯ БЕЗОПАСНОГО УДАЛЕНИЯ ФАЙЛА НУЖНО ИСПОЛЬЗОВАТЬ УТИЛИТУ SHRED

Использование rm не приводит к безопасному удалению файла, потому что остается возможность восстановления удаленного файла. При использовании утилиты shred файл удаляется более безопасным образом. Еще один метод безопасного удаления файла дан в описании утилиты dd в подразделе «Полное уничтожение файла».

Примеры

Следующая команда удаляет файлы как из рабочего каталога, так и из других каталогов:

```
$ rm memo
$ rm letter memo1 memo2
$ rm /home/sam/temp
```

В следующем примере перед удалением каждого файла в рабочем каталоге и его подкаталогах выдается запрос на подтверждение удаления:

```
$ rm -ir .
```

Эта команда применяется для удаления файлов, имена которых содержат специальные символы, особенно это касается пробелов, символов табуляции и символов разделителей строк. (Файлы с именами, содержащими такие символы, могут быть созданы совершенно случайно, не преднамеренным образом).

rmdir

Удаление каталогов.

*rmdir **список_каталогов***

Утилита rmdir удаляет пустые каталоги.

Аргументы

Список_каталогов — это список путевых имен пустых каталогов, удаляемых с помощью утилиты rmdir.

Ключи

При работе под управлением Linux утилита rmdir допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--ignore-fail-on-non-empty` Подавление вывода сообщения, которое rmdir обычно показывает при сбое в работе, вызванном тем, что каталог не пуст. С ключом `--parents` rmdir не завершает работу, если будет найден непустой каталог. (Linux)

`--parents -p` Удаление иерархии пустых каталогов.

`--verbose -v` Вывод имен каталогов по мере их удаления. (Linux)

Примечание

Если нужно удалить непустые каталоги вместе с их содержимым, следует воспользоваться утилитой rm с ключом `-r`.

Примеры

Следующая команда удаляет пустой каталог literature из рабочего каталога:

```
$ rmdir literature
```

Следующая команда удаляет каталог letters, используя абсолютное имя:

```
$ rmdir /home/sam/letters
```

Последняя команда удаляет каталоги letters, march и 05, при условии, что они пусты, за исключением других каталогов, указанных в пути:

```
$ rmdir -p letters/march/05
```

rsh

Выполнение команды на удаленной системе.

rsh [ключ] хост [командная_строка]

Утилита rsh запускает ***командную_строку*** на ***хосте*** путем запуска оболочки на удаленной системе. Без указания ***командной_строки*** rsh вызывает утилиту rlogin, которая осуществляет ваш вход на удаленную систему.

RSH НЕ ОБЕСПЕЧИВАЕТ ДОЛЖНУЮ СТЕПЕНЬ БЕЗОПАСНОСТИ

Для авторизации вашего входа в систему утилита rsh использует доверие на уровне хостов, которое не обеспечивает должную степень безопасности. Время от времени она отправляет ваш пароль по сети в виде простого текста, что противоречит нормам безопасности. Лучше по возможности пользоваться утилитой ssh (см. соответствующее описание).

Аргументы

Хост — это имя удаленной системы. Утилита rsh запускает ***командную_строку*** на удаленной системе. Специальные символы в ***командной_строке*** нужно процитировать, чтобы локальная оболочка их не расширила еще до передачи утилите rsh.

Ключи

`-l`*имя_пользователя* (login). Осуществление входа на удаленную систему под указанным *именем_пользователя*.

Примечания

Если в находящемся на удаленной системе файле по имени /etc/hosts.equiv указано имя локальной системы, удаленная система не станет выводить приглашение на ввод пароля. Системы, перечисленные в файле /etc/hosts.equiv, считаются такими же безопасными, как и локальная система.

Другой способ указания доверительных отношений базируется на индивидуальной для каждого пользователя основе. В личном каталоге каждого пользователя может содержаться файл по имени ~/.rhosts, в котором имеется список доверенных удаленных систем и пользователей. Образец файла .rhosts приведен в описании утилиты rlogin, в подразделе «Примеры».

Примеры

В первом примере пользователь Max использует утилиту rsh для получения списка файлов в своем личном каталоге на системе bravo:

```
$ rsh bravo ls
cost_of_living
info
preferences
work
```

Далее вывод предыдущей команды перенаправляется в файл bravo.ls. Поскольку символ перенаправления (>) не отключен, он интерпретируется локальной оболочкой, и файл bravo.ls создается на локальной системе.

```
$ rsh bravo ls > bravo_ls
$ cat bravo_ls
cost_of_living
info
preferences
work
```

В следующем примере символ перенаправления (>) берется в кавычки, поэтому файл bravo.ls создается на удаленной системе (bravo), как показывает команда ls, запущенная на системе bravo:

```
$ rsh bravo ls ">" bravo.ls
$ rsh bravo ls
bravo.ls
```

```
cost_of_living
info
preferences
work
```

В последнем примере утилита rsh, используемая без ***командной_строки***, осуществляет вход на удаленную систему. Пользователь Max воспользовался ключом `-l watson`, чтобы войти в систему bravo под именем `watson`. Файл ~watson/.rhosts должен быть настроен на разрешение пользователю Max входить в систему с использованием такой учетной записи. Образец файла .rhosts приведен в описании утилиты rlogin, в подразделе «Примеры».

```
$ rsh -l watson bravo
Last login: Sat Jul 30 16:13:53 from kudos
$ hostname
bravo
$ exit
rlogin: connection closed.   rlogin: подключение закрыто.
```

rsync

Безопасное копирование файлов и иерархий каталогов по сети.

rsync [ключи] [[пользователь@]с_хоста:]файл-источник [[пользователь @] на_хост:][файл-приемник]

Утилита rsync (remote synchronization — удаленная синхронизация) копирует обычные файлы или иерархию каталогов на локальной системе или с локальной системы на другую систему в сети или наоборот. По умолчанию для передачи файлов эта утилита использует OpenSSH и такой же механизм аутентификации, как в OpenSSH, поэтому она предоставляет такой же уровень безопасности, как и OpenSSH. Если нужно, утилита rsync выводит приглашение на ввод пароля. В качестве альтернативы можно использовать агентом передачи утилиту rsyncd, работающую в фоновом режиме.

Информация об утилите rsync дана в главе 14.

scp

Безопасное копирование одного или нескольких файлов на удаленную систему и из нее.

scp [[пользователь@]с_хоста:]файл-источник [[пользователь @]на_хост:] [файл-приемник]

Утилита scp (secure copy — безопасное копирование) копирует обычные файлы или файлы каталогов из одной системы в другую по сети. Для передачи файлов эта утилита использует протокол ssh и такой же механизм аутентификации, который используется в ssh; следовательно, она предоставляет такой же уровень безопасности, что и ssh. Если нужно, утилита scp выводит приглашение на ввод пароля.

Аргументы

Аргумент ***с_хоста*** является именем той системы, с которой копируются файлы, а аргумент ***на_хост*** — именем той системы, на которую они копируются. Если хост не указывать, scp предполагает, что нужно работать на локальной системе. В качестве аргумента ***пользователь*** на обеих системах по умолчанию используется имя пользователя, давшего команду на локальной системе; можно указать другое имя пользователя, заполнив часть командной строки ***пользователь@***. Утилита scp позволяет копировать файлы между двумя удаленными системами.

Файл-источник — это копируемый файл, а ***файл-приемник*** — получаемая копия. Можно указать обычный файл или файл каталога, использовав относительные или абсолютные путевые имена. Относительные путевые имена указываются относительно указанного или подразумеваемого личного каталога. Если ***файл-источник*** является каталогом, то чтобы копировать его содержимое, нужно использовать ключ `-r`. Когда ***файл-приемник*** является каталогом, каждый из ***файлов-источников*** сохраняет свое простое имя файла. Если ***файл-источник*** представлен только одним файлом, ***файл-приемник*** можно опустить, и скопированный файл будет иметь такое же простое имя файла, которое было у ***файла-источника***.

Ключи

`-P`*порт* Подключение к порту *порт* на удаленной системе. Учтите, что утилита ssh для указания порта использует ключ `-p` (с буквой в нижнем регистре).

`-p` (preserve — сохранение). Сохранение таких же как у исходного файла показателей времени последнего изменения и обращения, а также прав доступа.

`-q` (quiet — молчание). Отмена вывода указателя хода процесса.

`-r` (recursive — рекурсивный режим). Рекурсивное копирование иерархии каталога.

`-v` (verbose — подробно). Вывод отладочных сообщений о подключении и передаче данных. Этот ключ применяется в том случае, если что-то работает не так как ожидалось.

Примечания

Утилита scp принадлежит к набору инструментария безопасного сетевого подключения OpenSSH. Аспекты безопасности OpenSSH рассмотрены в описании утилиты ssh, в подразделе «Комментарии». Информация о сообщении, которое может быть получено при использовании утилиты scp для первоначального подключения к удаленной системе, дана в описании утилиты ssh, в подразделе «Первоначальная аутентификация».

Используя эту утилиту, можно копировать файлы с локальной системы и на локальную систему, а также между двумя удаленными системами. Нужно убедиться в наличии права на чтение копируемого файла и права на запись в тот каталог, в который копируется файл.

Групповые символы в удаленных путевых именах (такие как *) нужно цитировать, чтобы они интерпретировались оболочкой на удаленной, а не на локальной системе.

Как и при использовании утилиты cp, если ***файл-приемник*** существует до запуска scp, эта утилита его перепишет.

Примеры

В первом примере файлы с именами, оканчивающимися на .c, копируются из рабочего каталога на локальной системе в каталог ~sam/archives на системе bravo. Групповой символ не процитирован, поэтому оболочка локальной системы его расширит. Поскольку для каталога archives указано относительное имя, scp предполагает, что он является подкаталогом личного каталога пользователя Sam на системе bravo. Каждый копируемый файл сохраняет свое простое имя файла.

```
$ scp *.c sam@bravo:archives
```

Затем пользователь Max копирует структуру каталога ниже ~max/memos на системе по имени bravo в каталог ~sam/max.memos.bravo на системе по имени kudos. У него должно быть необходимое право на запись в личный каталог пользователя Samна в системе kudos.

```
$ scp -r bravo:memos sam@kudos:max.memos.bravo
```

В завершение пользователь Max копирует файлы с именами, оканчивающимися на .c, из принадлежащего пользователю Sam каталога archives на системе bravo в каталог sam.c.bravo в своем рабочем каталоге. Групповой символ процитирован, чтобы уберечь его от расширения локальной оболочкой; он будет интерпретирован удаленной системой bravo.

```
$ scp -r 'sam@bravo:archives/*.c' sam.c.bravo
```

При копировании нескольких файлов или каталогов приемник, будь то на локальной или удаленной системе, должен быть существующим каталогом, а не обычным или несуществующим файлом.

sed

Редактирование файла в неинтерактивном режиме.

sed [–n] ***программа [список_файлов]***
sed [–n] –f ***файл_программы [список_файлов]***

Утилита sed (stream editor — потоковый редактор) является пакетным (неинтерактивным) редактором. Она преобразует вводимый поток, который может поступать из файла или из стандартного ввода. Эта утилита часто используется в канале в качестве фильтра. Поскольку она осуществляет только один проход вводимой информации, утилита sed является более эффективным средством по сравнению с таким интерактивным редактором, как ed. Со многими дистрибутивами Linux поставляется утилита GNU sed, а в системе Mac OS X предоставляется утилита BSD sed. Материалы главы 13 могут применяться к обеим версиям этой утилиты.

Информация об утилите sed дана в главе 13.

setFile (OS X)

Установка атрибутов файла.

SetFile [ключи] список_файлов

Утилита SetFile устанавливает атрибуты файла, включая коды его типа и создателя, время его создания и последнего изменения и флаги атрибутов, такие как флаги невидимости и заблокированности.

Аргументы

Список_файлов указывает путевые имена одного или нескольких файлов, с которыми работает утилита SetFile.

Ключи

Ключи для утилиты SetFile соответствуют ключам для утилиты GetFileInfo (см. описание этой утилиты).

`-a` *флаги* (attribute — атрибут). Установка указанных *флагов* атрибутов. Буква верхнего регистра для указания флага приводит к его установке, а буква нижнего регистра — к снятию. Значения неуказанных *флагов* не изменяются. Список флагов атрибутов дан на man-странице, посвященной утилите SetFile.

`-c` *создатель* Установка для кода создателя значения *создатель*.

`-d` *дата* Установка *даты* создания файла. *Дата* должна иметь формат `мм/дд/[rr]rr [чч:мм:[:сс] [AM | PM]]`. Если не указать `AM` или `PM` (до или после полудня), SetFile предположит, что время задано в 24-часовом формате. Строка *дата*, содержащая пробелы, должна быть заключена в кавычки.

`-m` *дата* (modification — изменение). Установка *даты* последнего изменения файла. Используется такой же формат даты, как и для ключа `-d`.

`-P` (без разыменования). Для каждого файла, являющегося символьной ссылкой, установка информации о символьной ссылке, а не о файле, на который она указывает. Действие этого ключа распространяется на все файлы. При его использовании файлы, не являющиеся символьными ссылками, обрабатываются обычным образом. Пример использования этого ключа приведен в разделе «chgrp», в подразделе «Примеры».

`-t` *тип* Установка значения *тип* для кода типа файла.

Примечания

Утилита SetFile является частью дополнительно устанавливаемого инструментария разработчика — Developer Tools. Ключи для SetFile и соответствующие им ключи для GetFileInfo имеют небольшие отличия. Например, для ключа `-a` при использовании утилиты SetFile можно указать несколько флагов атрибутов, а при использовании утилиты GetFileInfo только один. Кроме того, для утилиты SetFile требуется наличие

пробела между ключом -a и перечнем флагов, а утилита GetFileInfo в этом месте пробела не допускает.

Примеры

В первом примере устанавливаются коды типа и создателя файла по имени arch в SIT5 и SIT! соответственно, показывая, что этот файл является архивом StuffIt. Эти коды показывает утилита GetFileInfo.

```
$ SetFile -t SIT5 -c SIT! arch
$ GetFileInfo -c arch
"SIT!"
$ GetFileInfo -t arch
"SIT5"
```

В следующем примере файл по имени secret помечается невидимым и заблокированным. Этот файл не будет виден в Finder, и большинство приложений OS X не смогут его переписать.

```
$ SetFile -a VL secret
```

В последнем примере для каждого файла, присутствующего в рабочем каталоге, снимается флаг атрибута невидимости:

```
$ SetFile -a v *
```

sleep

Создание процесса, приостановленного на указанный интервал времени.

sleep ***время***
sleep ***список_времен*** (Linux)

Утилита sleep заставляет вызываемый процесс приостанавливать свое выполнение на указанное время.

Аргументы

Традиционно количество времени, в течение которого процесс находится в приостановленном состоянии, задается в виде одного целочисленного аргумента, ***времени***, определяющего количество секунд. Но ***время*** не обязательно должно быть целым числом: можно указать и десятичную часть числа. При работе под управлением Linux можно также добавить и указание на единицу измерения времени: s (секунды), m (минуты), h (часы) или d (дни).

При работе под управлением Linux можно составить ***список_времен***, включая в командную строку несколько показателей времени: общее время пребывания процесса в приостановленном состоянии будет складываться из этих времен. Например, если указать 1h 30m 100s, процесс будет пребывать в «спячке» 91 минуту и 40 секунд.

Примеры

Утилиту sleep можно использовать из командной строки для выполнения команды после определенного периода времени. В следующем примере выполняется фоновый процесс, напоминающий о необходимости позвонить по телефону через 20 минут (1200 секунд):

```
$ (sleep 1200; echo "Напоминание о звонке.") &
[1] 4660
```

Кроме этого под управлением Linux для точно такого же напоминания можно дать следующую команду:

```
$ (sleep 20m; echo "Напоминание о звонке.") &
[2] 4667
```

Утилиту sleep можно также использовать в сценарии оболочки для выполнения команды через постоянные интервалы времени. Например, сценарий оболочки per выполняет программу по имени update каждые 90 секунд:

```
$ cat per
#!/bin/bash
while true
do
    update
    sleep 90
done
```

Если выполнять такой сценарий оболочки, как per, в фоновом режиме, его можно будет завершить только с помощью утилиты kill.

Последний сценарий оболочки принимает в качестве аргумента имя файла и ждет, пока этот файл не появится на диске. Если файл не существует, сценарий приостанавливает свою работу на одну минуту и 45 секунд, после чего опять проверяет, не появился ли файл.

```
$ cat wait_for_file
#!/bin/bash

if [ $# != 1 ]; then
    echo "Usage: wait_for_file filename"
    exit 1
fi

while true
do
    if [ -f "$1" ]; then
        echo "$1 is here now"
        exit 0
    fi
    sleep 1m 45
done
```

При работе под управлением Mac OS X нужно заменить `1m 45` на `105`.

sort

Сортировка и (или) объединение файлов.

sort [ключи] [список_файлов]

Утилита sort сортирует и (или) объединяет один или несколько текстовых файлов.

Аргументы

Список_файлов — это список путевых имен, состоящий из одного или нескольких имен обычных текстовых файлов. Если ***список_файлов*** опущен, sort получает свой ввод со стандартного ввода. Без ключа `-o` sort отправляет свой вывод на стандартный вывод. Эта утилита сортирует и объединяет файлы, если только не использован ключ –m (только объединение) или ключ –c (только проверка).

Ключи

Когда ключ не указан, sort сортирует файл в машинной последовательности упорядочивания (обычно в соответствии с порядком следования ASCII-символов). Без ключа `--key` sort сортирует файл построчно. Чтобы указать поля сортировки внутри строки, нужно использовать ключ `--key`. За ключом `--key` могут следовать дополнительные ключи, имеющие ведущий дефис; дополнительные сведения даны в подразделе «Комментарии».

ВЕРСИЯ SORT ДЛЯ MAC OS X ДОПУСКАЕТ ПРИМЕНЕНИЕ ДЛИННЫХ КЛЮЧЕЙ

Ключи для утилиты sort, перед которыми ставится двойной дефис (– –), работают как под управлением Mac OS X, так и под управлением Linux.

`--ignore-leading-blanks` `-b` Пробельные символы (символы табуляции и пробелы) обычно являются меткой начала полей в файле ввода. Без этого ключа sort воспринимает ведущие пробелы как часть того поля, которому они предшествуют. Этот ключ заставляет игнорировать ведущие пробельные символы в поле, и sort не воспринимает их при проведении сравнений в процессе сортировки.

`--check` `-c` Проверка того, что файл отсортирован должным образом. Если все в порядке, утилита sort не выводит никакой информации. Если файл не был отсортирован, утилита выводит сообщение и возвращает статус 1.

`--dictionary-order` `-d` Игнорирование всех символов, не являющихся алфавитно-цифровыми или пробельными. Например, с этим ключом sort не берет в расчет знаки пунктуации.

`--ignore-case` `-f` (fold — принимать за). Прием всех букв в нижнем регистре за буквы в верхнем регистре. Этот ключ следует использовать при сортировке файла, который содержит буквы как в верхнем, так и в нижнем регистре.

`--ignore-nonprinting` `-i` Игнорирование символов, не выводимых на печать. Действие этого ключа отменяется ключом `--dictionary-order`.

`--key=`*старт*`[,`*стоп*`]`
`-k` *старт*`[,`*стоп*`]`
Указание полей сортировки внутри строки. Без этого ключа используется полнострочный порядок сортировки. Поле сортировки начинается в позиции строки, указанной аргументом *старт*, и заканчивается на позиции, указанной аргументом *стоп*, или, если *стоп* опущен, в конце строки. Позиции *старт* и *стоп* указываются в формате *f[.c]*, где *f* — это номер поля, а *c* — необязательный символ в этом поле. Нумерация начинается с единицы. Если *c* в аргументе *старт* не указан, по умолчанию им становится первый символ в поле; когда *c* не указан в аргументе *стоп*, по умолчанию им становится последний символ в этом поле. Дополнительные объяснения, касающиеся полей сортировки, даны в подразделе «Комментарии», а иллюстрации их использования — в подразделе «Примеры».

`--merge -m`
Предположение о том, что каждый из нескольких файлов ввода уже отсортирован, и объединение их без проверки сортировки.

`--numeric-sort -n`
Сортировка в арифметическом порядке; отмена сортировки строк или полей в машинной последовательности упорядочивания. При использовании этого ключа знак минус и десятичная точка приобретают свое арифметическое значение.

`--output=`*имя_файла*
`-o` *имя_файла*
Отправка вывода не на стандартный вывод, а в файл по имени *имя_файла*, которое должно быть таким же, как одно из имен в ***списке_файлов***.

`--reverse -r`
Инвертирование смысла сортировки (например, `z` идет перед `a`).

`--field-separator=`*x*
`-t` *x*
Указание *x* в качестве разделителя полей. Дополнительные сведения о разделителях полей даны в подразделе «Комментарии».

`--unique -u`
Вывод повторяющихся строк только один раз. При использовании этого ключа с ключом `--check` утилита sort выводит сообщение, если одна и та же строка появляется в файле ввода более одного раза, даже если файл находится в отсортированном состоянии.

Комментарий

Без ключей утилита sort осуществляет построчное упорядочивание.

В следующем описании под *полем* понимается последовательность символов в строке ввода. Без ключа `--field-separator` поля ограничены пустой строкой, предшествующей группе из одного или нескольких пробельных символов (символов табуляции или пробелов). Увидеть пустые строки, являющиеся границами полей, невозможно, потому что это воображаемая точка между двумя полями. Поля также ограничены началом и концом строки. Строка, показанная на рис. V.1, содержит поля `Toni`, `ПРОБЕЛBarnett` и `РОБЕЛПРОБЕЛПРОБЕЛПРОБЕЛ55020`. Эти поля определяют поля сортировки. Иногда поля и поля сортировки совпадают.

Поле сортировки
Поле сортировки — это последовательность символов, которая используется утилитой sort, чтобы выстроит строки по порядку. Поле сортировки может содержать все содержимое или часть одного или нескольких полей (см. рис. V.1).

Ключ `--key` содержит в качестве аргументов пары указателей, определяющих подразделы каждой строки (поля сортировки), используемые для сравнений. Подробности даны в описании ключа `--key`.

Лидирующие пустые места Ключ `-b` заставляет утилиту sort проигнорировать ведущие пробельные символы в поле сортировки. Если этот ключ не используется, sort рассматривает каждый пробельный символ как символ в поле сортировки и включает его в сравнение, проводимое при сортировке.

Ключи Можно указать ключи, которые относятся только к заданному полю сортировки, поставив их сразу же после *стоп*-указателя (или *старт*-указателя, если *стоп*-указатель отсутствует), с одним из ключей `b`, `d`, `f`, `i`, `n` или `r`. В этом случае ставить дефис перед ключом не нужно.

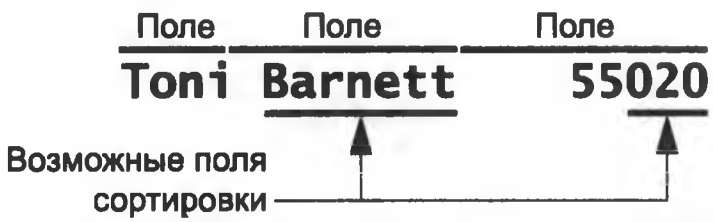

Рис. V.1. Поля и поля сортировки

Несколько полей сортировки Когда указано более одного поля сортировки, утилита sort рассматривает их в том порядке, в котором они указаны в командной строке. Если первое поле сортировки двух строк оказывается одинаковым, sort рассматривает второе поле сортировки. Если и оно оказывается одинаковым, sort рассматривает третье поле. Этот процесс продолжается для всех указанных полей сортировки. Если все поля сортировки оказываются одинаковыми, sort рассматривает всю строку целиком.

Примеры

Примеры в этом разделе показывают некоторые свойства и приемы использования утилиты sort. Примеры предполагают наличие в рабочем каталоге следующего файла по имени list:

```
$ cat list
Tom Winstrom        94201
Janet Dempsey       94111
Alice MacLeod       94114
David Mack          94114
Toni Barnett        95020
Jack Cooper         94072
Richard MacDonald   95510
```

Этот файл содержит список имен и ZIP-кодов (почтовых индексов). Каждая строка файла содержит три поля: имени, фамилии и ZIP-кода. Чтобы примеры работали, в качестве пробельных символов в файле должны использоваться пробелы, а не символы табуляции.

В первом примере показывается работа утилиты sort без использования ключей, единственным аргументом служит имя файла ввода. В этом случае sort проводит построчную сортировку содержимого файла. При одном и том же первом символе на двух строках sort для определения правильного порядка следования строк рассматривает вторые символы. Если одинаковы и вторые символы, sort рассматривает третьи символы. Этот процесс продолжается до тех пор, пока sort не обнаружит разные символы в строках. Если строки идентичны друг другу, то какую из них sort поставит первой, не имеет значения. В данном примере утилите в большинстве случаев нужно проверять только первые три символа каждой строки. Она выводит список, выстроенный в алфавитном порядке по имени.

```
$ sort list
Alice MacLeod        94114
David Mack           94114
Jack Cooper          94072
Janet Dempsey        94111
Richard MacDonald    95510
Tom Winstrom         94201
Toni Barnett         95020
```

Можно указать утилите sort перед началом сравнивания строк пропустить любое количество полей или символов в строке. Пробельные символы обычно являются признаком начала поля. В следующем примере тот же самый список сортируется по фамилиям, то есть по второму полю: аргумент `--key=2` заставляет утилиту sort начинать сравнение с этого поля. Поскольку второй указатель отсутствует, поле сортировки распространяется до конца строки. Теперь список практически отсортирован по фамилиям, но есть проблема со строками Mac.

```
$ sort --key=2 list
Toni Barnett         95020
Jack Cooper          94072
Janet Dempsey        94111
Richard MacDonald    95510
Alice MacLeod        94114
David Mack           94114
Tom Winstrom         94201
```

В предыдущем примере MacLeod был выше Mack. После определения, что поля сортировки этих двух строк имеют одинаковое значение вплоть до третьей буквы (Mac), утилита sort помещает L перед k, поскольку выстраивание строк ведется на основе кодов ASCII-символов, где буквы верхнего регистра идут перед буквами нижнего регистра.

Ключ `--ignore-case` заставляет утилиту sort рассматривать буквы верхнего и нижнего регистров одинаковым образом, что устраняет проблему с MacLeod и Mack:

```
$ sort --ignore-case --key=2 list
Toni Barnett         95020
Jack Cooper          94072
Janet Dempsey        94111
Richard MacDonald    95510
David Mack           94114
```

```
Alice MacLeod        94114
Tom Winstrom         94201
```

В следующем примере предпринимается попытка отсортировать список по третьему полю, то есть по ZIP-коду. В этом случает sort не помещает номера по порядку, а вместо этого в отсортированном списке пускает более короткие имена первыми, а более длинные — последними. Аргумент `--key=3` заставляет утилиту sort начинать сравнение с третьего поля, то есть с ZIP-кода. Поле начинается с пробельного символа и включает последующие пробельные символы. В случае с файлом list в качестве пробельных символов выступают пробелы. ASCII-значение у пробела меньше, чем у любого выводимого на печать символа, поэтому sort помещает ZIP-код, которому предшествует наибольшее количество пробелов, первым, а ZIP-код, которому предшествует наименьшее количество пробелов, последним.

```
$ sort --key=3 list
David Mack           94114
Jack Cooper          94072
Tom Winstrom         94201
Toni Barnett         95020
Janet Dempsey        94111
Alice MacLeod        94114
Richard MacDonald    95510
```

Ключ `-b` (`--ignore-leading-blanks`) заставляет утилиту sort игнорировать имеющиеся в поле ведущие пробелы. При использовании этого ключа ZIP-коды выходят после сравнения в правильном порядке. Когда утилита sort определяет, что у `MacLeod` и `Mack` одинаковые ZIP-коды, она сравнивает все строки, помещая `Alice MacLeod` перед `David Mack` (потому что `A` идет раньше `D`).

```
$ sort -b --key=3 list
Jack Cooper          94072
Janet Dempsey        94111
Alice MacLeod        94114
David Mack           94114
Tom Winstrom         94201
Toni Barnett         95020
Richard MacDonald    95510
```

Для сортировки по фамилии в алфавитном порядке при одинаковых ZIP-кодах утилите sort нужно осуществить повторный проход, который отсортирует список по полю фамилии. В следующем примере показано, как осуществить этот второй проход путем указания второго поля сортировки и использования ключа `-f` (`--ignore-case`), чтобы предотвратить повторное возникновение проблемы с фамилиями `Mack` и `MacLeod`:

```
$ sort -b -f --key=3 --key=2 list
Jack Cooper          94072
Janet Dempsey        94111
David Mack           94114
Alice MacLeod        94114
Tom Winstrom         94201
Toni Barnett         95020
Richard MacDonald    95510
```

В следующем примере показывается команда **sort**, которая пропускает не только поля, но и символы. Ключ `-k 3.4` (являющийся эквивалентом ключа `--key=3.4`) заставляет **sort** начинать сравнение с четвертого символа третьего поля. Поскольку команда не определяет, где кончается поле сортировки, по умолчанию концом поля определяется конец строки. Полем сортировки становятся последние две цифры ZIP-кода.

```
$ sort -fb -k 3.4 list
Tom Winstrom          94201
Richard MacDonald     95510
Janet Dempsey         94111
Alice MacLeod         94114
David Mack            94114
Toni Barnett          95020
Jack Cooper           94072
```

Проблема сортировки по фамилии внутри сортировки по последним двум цифрам ZIP-кода решается путем второго прохода, охватывающего поле фамилии. Ключ `f`, который следует за `-k 2`, влияет на второй проход, который выстраивает строки только по фамилии.

```
$ sort -b -k 3.4 -k 2f list
Tom Winstrom          94201
Richard MacDonald     95510
Janet Dempsey         94111
David Mack            94114
Alice MacLeod         94114
Toni Barnett          95020
Jack Cooper           94072
```

Следующий набор примеров использует файл данных **cars**. Слева направо столбцы в этом файле содержат изготовителя каждой машины, модель, год выпуска, пробег и цену:

```
$ cat cars
plym    fury    1970    73      2500
chevy   malibu  1999    60      3000
ford    mustang 1965    45      10000
volvo   s80     1998    102     9850
ford    thundbd 2003    15      10500
chevy   malibu  2000    50      3500
bmw     325i    1985    115     450
honda   accord  2001    30      6000
ford    taurus  2004    10      17000
toyota  rav4    2002    180     750
chevy   impala  1985    85      1550
ford    explor  2003    25      9500
```

Без использования ключей утилита **sort** выводит отсортированную копию файла:

```
$ sort cars
bmw     325i    1985    115     450
chevy   impala  1985    85      1550
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
```

```
ford    explor  2003    25      9500
ford    mustang 1965    45      10000
ford    taurus  2004    10      17000
ford    thundbd 2003    15      10500
honda   accord  2001    30      6000
plym    fury    1970    73      2500
toyota  rav4    2002    180     750
volvo   s80     1998    102     9850
```

Целью следующего примера станет сортировка по производителю и по цене внутри производителя. Пока не будет указано что-либо иное, поле сортировки распространяется до конца строки. Ключ `-k 1` задает поле сортировки с начала строки. Командная строка заставляет утилиту **sort** вести сортировку по всей строке, а затем совершить второй проход, сортируя по пятому полю все строки, чьи поля сортировки при первом проходе были одинаковыми (`-k 5`):

```
$ sort -k 1 -k 5 cars
bmw     325i    1985    115     450
chevy   impala  1985    85      1550
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
ford    explor  2003    25      9500
ford    mustang 1965    45      10000
ford    taurus  2004    10      17000
ford    thundbd 2003    15      10500
honda   accord  2001    30      6000
plym    fury    1970    73      2500
toyota  rav4    2002    180     750
volvo   s80     1998    102     9850
```

Поскольку двух одинаковых полей не было, утилита **sort** делает только один проход, проводя сортировку по каждой полной строке. (Если две строки различаются только по пятому полю, они все равно будут правильно отсортированы и при первом проходе, поэтому второй проход не нужен.) Посмотрите на строки, содержащие `taurus` и `thundbd`. Они отсортированы по второму, а не по пятому полю, показывая тем самым, что эта сортировка никогда не будет проводить второй проход и, следовательно, никогда не будет сортировать по пятому полю.

В следующем примере сортировка при первом проходе вынуждена останавливаться в конце первого поля. Ключ `-k 1,1` определяет стартовый указатель на первом символе первого поля, а стоповый указатель — на последнем символе первого поля. Если не указывается символ внутри стартового указателя, то по умолчанию указатель устанавливается на первый символ, а если не указывается символ внутри стопового указателя, то по умолчанию указатель устанавливается на последний символ. Теперь `taurus` и `thundbd` будут правильно отсортированы по цене. Но посмотрите на `explor`: он стоит меньше, чем другие автомобили Ford, но утилита **sort** спозиционировала его как более дорогостоящий. Утилита **sort** расположила список по порядку следования ASCII-кодов, а не в числовом порядке: 9500 следует после 10000, поскольку 9 следует после 1.

```
$ sort -k 1,1 -k 5 cars
bmw     325i    1985    115     450
chevy   impala  1985    85      1550
chevy   malibu  1999    60      3000
```

```
chevy   malibu  2000    50      3500
ford    mustang 1965    45      10000
ford    thundbd 2003    15      10500
ford    taurus  2004    10      17000
ford    explor  2003    25      9500
honda   accord  2001    30      6000
plym    fury    1970    73      2500
toyota  rav4    2002    180     750
volvo   s80     1998    102     9850
```

Ключ `-n` (числовой) во втором проходе располагает список в правильном порядке:

```
$ sort -k 1,1 -k 5n cars
bmw     325i    1985    115     450
chevy   impala  1985    85      1550
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
ford    explor  2003    25      9500
ford    mustang 1965    45      10000
ford    thundbd 2003    15      10500
ford    taurus  2004    10      17000
honda   accord  2001    30      6000
plym    fury    1970    73      2500
toyota  rav4    2002    180     750
volvo   s80     1998    102     9850
```

В следующем примере опять показывается, что пока не будет указано что-либо иное, утилита `sort` выстраивает содержимое файла, начиная сравнение с указанного поля и продолжая его до конца строки. Она не проводит второй проход, пока два первых поля сортировки не будут одинаковыми. Поскольку спецификатор первого поля сортировки не имеет стопового указателя, поле сортировки для первого прохода включает третье поле и продолжается до самого конца строки. Несмотря на то что этот пример сортирует машины по годам, он не сортирует их по моделям внутри производителей и внутри годов выпуска (`ford thundbd` следует перед `ford explor`, а эти строки должны следовать в обратном порядке).

```
$ sort -k 3 -k 1 cars
ford    mustang 1965    45      10000
plym    fury    1970    73      2500
bmw     325i    1985    115     450
chevy   impala  1985    85      1550
volvo   s80     1998    102     9850
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
honda   accord  2001    30      6000
toyota  rav4    2002    180     750
ford    thundbd 2003    15      10500
ford    explor  2003    25      9500
ford    taurus  2004    10      17000
```

Указывая окончание поля сортировки для первого прохода, позволяет утилите **sort** правильно провести свой второй проход:

```
$ sort -k 3,3 -k 1 cars
ford    mustang 1965    45      10000
```

```
plym    fury    1970    73      2500
bmw     325i    1985    115     450
chevy   impala  1985    85      1550
volvo   s80     1998    102     9850
chevy   malibu  1999    60      3000
chevy   malibu  2000    50      3500
honda   accord  2001    30      6000
toyota  rav4    2002    180     750
ford    explor  2003    25      9500
ford    thundbd 2003    15      10500
ford    taurus  2004    10      17000
```

В следующих примерах показываются важные технологии сортировки: выстраивание списка в алфавитном порядке, объединение записей в верхнем и нижнем регистрах и устранение дубликатов. Для этого используется следующий неотсортированный список:

```
$ cat short
Pear
Pear
apple
pear
Apple
```

Далее показана простая сортировка:

```
$ sort short
Apple
Pear
Pear
apple
pear
```

Результат следующей сортировки, при которой буквы нижнего регистра принимаются за буквы верхнего регистра, для начала кажется неплохим, но он не избавлен от дубликатов:

```
$ sort -f short
Apple
apple
Pear
Pear
pear
```

Ключ `-u` (`unique` — уникальный) исключает дубликаты, но без ключа `-f` сначала следуют записи в верхнем регистре:

```
$ sort -u short
Apple
Pear
apple
pear
```

При попытке использования обоих ключей `-u` и `-f` некоторые записи теряются:

```
$ sort -uf short
apple
Pear
```

Решение кроется в двух проходах. При обоих проходах проводится уникальная сортировка, и при первом проходе буквы нижнего регистра принимаются за буквы верхнего регистра:

```
$ sort -u -k 1f -k 1 short
Apple
apple
Pear
pear
```

split

Разбиение файла на части.

split [ключи] [имя_файла [префикс]]

Утилита split разбивает свой ввод на части размером 1000 строк, которые называются xaa, xab, xac и т. д. Последняя часть может быть короче остальных. Ключи могут изменять размер частей и длину имен.

Аргументы

Имя_файла — это полное имя файла, обрабатываемого утилитой split. Если аргумент на указан или если вместо ***имени_файла*** указан дефис (-), split читает данные со стандартного ввода. ***Префикс*** состоит из одного или нескольких символов, которые утилита split использует в качестве префиксов имен создаваемых файлов, по умолчанию в качестве префикса используется буква x.

Ключи

При работе под управлением Linux утилита split допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--suffix-length=`*длина*
`-a` *длина*
: Указание *длины* суффикса имени файла в символах (по умолчанию *длина* равна 2).

`--bytes=`*n*`[`*u*`]`
`-b` *n*`[`*u*`]`
: Разбиение ввода на файлы длиной *n* байт. Аргумент *u* является дополнительной единицей измерения, которая может принимать значения `k` (килобайт или 1024-байтовые блоки) или `m` (мегабайт или 1 048 576-байтовые блоки). Если включить в аргументы единицу измерения, утилита split использует не байты, а указанные единицы. При работе под управлением Linux *u* может иметь значение `b` (512-байтовые блоки) или быть любым из суффиксов, перечисленных в табл. V.1.

`--numeric-suffixes`
`-d`
: Указание числовых суффиксов вместо буквенных.

`--lines=`***количество***
`num -l`
количество

Разбиение ввода на файлы длиной в указанное *количество* строк (по умолчанию *количество* равно 1000).

Комментарии

По умолчанию утилита split дает первому создаваемому файлу имя xaa. Буква x является префиксом по умолчанию. Префикс можно изменить с помощью аргумента командной строки ***префикс***. Можно изменить количество символов каждого имени файла, указав за ***префиксом*** ключ `--suffix-length`.

Примеры

По умолчанию split разбивает файл на части длиной в 1000 строк с именами xaa, xab, xac и т. д. Утилита wc с ключом `-l` показывает количество строк в каждом файле. Последний файл по имени xar имеет длину меньше, чем у остальных файлов.

```
$ split /usr/share/dict/words
$ wc -l *
  1000 xaa
  1000 xab
  1000 xac
…
  1000 xdt
   569 xdu
 98569 total
```

В следующем примере для указания строки SEC в качестве префикса имен файлов используется аргумент ***префикс***, а для изменения количества букв в суффиксе имени файла и использования суффикса из трех букв используется ключ `-c` (`--suffix-length`):

```
$ split -a 3 /usr/share/dict/words SEC
$ ls
SECaaa  SECaak  SECaau  SECabe  SECabo  SECaby  SECaci  SECacs  SECadc  SECadm
SECaab  SECaal  SECaav  SECabf  SECabp  SECabz  SECacj  SECact  SECadd  SECadn
…
SECaaj  SECaat  SECabd  SECabn  SECabx  SECach  SECacr  SECadb  SECadl
```

ssh

Безопасное выполнение команд на удаленной системе.

ssh [ключ] [пользователь@]хост [командная_строка]

Утилита ssh осуществляет вход на удаленную систему и запускает оболочку. Кроме этого ssh выполняет команду на удаленной системе и осуществляет выход из системы. Утилита ssh, которая может заменить собой утилиты rsh и rlogin, предоставляет безопасный, закодированный обмен данными между двумя системами по небезопасной сети.

Аргументы

Хост — это имя системы, в которую нужно войти или на которой нужно запустить команду. Если не пройден один из нескольких видов аутентификации, утилита ssh выводит приглашение на ввод имени пользователя и пароля для удаленной системы. Когда ssh имеет возможность автоматического входа в систему, она делает это от имени пользователя, запустившего команду ssh, или от имени пользователя, указанного в аргументе ***пользователь***@ в командной строке ssh.

Командная_строка запускается на удаленной системе. Если ***командная_строка*** не используется, утилита ssh осуществляет вход на удаленную систему. Если не нужно, чтобы локальная оболочка расширила имеющиеся в ***командной_строке*** специальные символы, они должны быть процитированы.

Ключи

`-l` *пользователь* (login — вход в систему). Попытка входа в систему под именем ***пользователь***. Этот ключ является эквивалентом использования аргумента ***пользователь***@ в командной строке.

`-n` (null). Перенаправление стандартного ввода утилиты ssh на получение его с устройства /dev/null. См. описание ключа `-f`.

`-p` *порт* Подключение к порту ***порт*** на удаленной системе. Следует иметь в виду, что утилита scp использует для указания порта ключ `-P` (буква в верхнем регистре).

`-q` (quiet — молчание). Подавление предупреждающих и диагностических сообщений.

`-t` (tty). Выделение утилитой ssh псевдо-tty процессу на удаленной системе. Без этого ключа при запуске команды на удаленной системе ssh не выделяет процессу tty (терминал). Вместо этого утилита ssh подключает к удаленному процессу на время своего сеанса работы стандартный ввод и стандартный вывод, что вполне в порядке вещей, но не всегда является желаемым режимом работы. Этот ключ заставляет ssh выделить tty на удаленной системе, чтобы работали программы, требующие применения tty.

`-v` (verbose — подробно). Вывод отладочных сообщений о подключении и передаче данных. Этот ключ применяется в том случае, если что-то работает не так как ожидалось.

`-X` (X11). Включение режима X11 forwarding. Этот ключ может и не понадобиться, поскольку режим X11 forwarding может быть включен в конфигурационном файле.

`-x` (X11). Выключение режима X11 forwarding.

Комментарии

OpenSSH Используя шифрование открытым ключом, OpenSSH предоставляет два уровня аутентификации: сервера и клиента (пользователя). Сначала клиент

проверяет, что подключился к правильному серверу. Затем OpenSSH шифрует связь между системами. После установки безопасного зашифрованного соединения OpenSSH убеждается, что у пользователя есть право на вход в систему и на копирование файлов с сервера и на сервер. После проверки сервера и пользователя OpenSSH может разрешить различным службам проводить свою работу через подключение, включая интерактивные сессии оболочки (ssh), удаленное выполнение команд (ssh), копирование файлов (scp), использование FTP-служб (sftp), клиент-серверные X11-соединения и TCP/IP туннелирование порта.

ssh Утилита ssh позволяет войти на удаленную систему по сети. Например, можно выбрать использование удаленной системы для доступа к узкоспециализированному приложению или попользоваться устройством, которое есть только на этой системе, или воспользоваться удаленной системой из-за того, что она быстрее или меньше загружена, чем локальная система. Во время путешествия многие бизнесмены используют ssh на ноутбуках для входа в систему, работающую в офисе компании. В среде графического интерфейса пользователя можно одновременно использовать для входа сразу несколько систем, задействовав для каждой из них отдельное окно эмулятора терминала.

X11 forwarding Имея включенный надежный режим X11 forwarding, можно запросто запустить по ssh-соединению X11-программу: запустите ssh из эмулятора терминала, работающего на X11-сервере, и дайте X11 такую команду, как xclock; графический вывод появится на локальном дисплее. Для включения режима X11 forwarding установите в конфигурационном файле /etc/ssh/ssh_config значение для переменной ForwardX11Trusted в yes. Чтобы повысить уровень безопасности (и в ряде случаев понизить уровень удобства работы), установите для ForwardX11Trusted значение no.

known_hosts, ssh_known_hosts Позитивно идентифицируемые подключаемые хосты локальной системы перечисляются в двух файлах: ~/.ssh/known_hosts (пользовательский) и /etc/ssh/ssh_known_hosts (глобальный). Никто, кроме владельца (пользователя с root-правами в случае со вторым файлом), не может вести запись ни в один из этих файлов. Никто, кроме владельца, не может иметь доступ к каталогу ~/.ssh.

Первоначальная аутентификация Когда подключение к OpenSSH-серверу проводится в первый раз, OpenSSH-клиент выводит приглашение на подтверждение, что вы подключились к правильной системе. Эта проверка может помочь избежать атак, организуемых неким злоумышленником, перехватывающим и имитирующим обмен сообщениями (man-in-the-middle attack):

```
The authenticity of host 'plum (192.168.0.10)' can't be established.
        Аутентификация хоста 'plum (192.168.0.10)' невозможна
RSA key fingerprint is d1:9d:1b:5b:97:5c:80:e9:4b:41:9a:b7:bc:1a:ea:a1.
        Отпечаток RSA-ключа d1:9d:1b:5b:97:5c:80:e9:4b:41:9a:b7:bc:1a:ea:a1.
Are you sure you want to continue connecting (yes/no)? yes
        Продолжить подключение (да/нет)? да
Warning: Permanently added 'plum,192.168.0.10' (RSA) to the list of
known hosts.
        Предупреждение: Теперь 'plum,192.168.0.10' (RSA) добавлен в постоянный
        список известных хостов.
```

Перед тем как ответить на предыдущий запрос, убедитесь в том, что подключились к правильной системе, а не попались в ловушку обманщика. Если есть сомнения, позвоните кому-нибудь, кто вошел в эту систему по локальной сети и сможет помочь проверить факт вашего обращения к правильной системе. После ответа «yes» (нужно ввести все слово целиком) клиент добавляет серверный открытый хост-ключ (одну строку файла /etc/ssh/ssh_host_rsa_key.pub или файла /etc/ssh/ssh_host_dsa_key.pub на сервере) к пользовательскому файлу ~/.ssh/known_hosts на локальной системе, создавая, если нужно, каталог ~/.ssh. Чтобы обеспечить возможность отслеживания принадлежности строк в файле known_hosts определенному серверу, OpenSSH ставит на первое место в строке имя сервера и его IP-адрес.

При последующем использовании OpenSSH для подключения к этому серверу клиент проверяет факт подключения к нужному серверу путем сравнения этого ключа с одним из ключей, предоставляемых сервером.

После того как открытый ключ удаленной системы сохранен в одном из файлов known-hosts, при каждой попытке удаленной системы предоставить другой отпечаток при соединении систем OpenSSH выводит следующее сообщение и отказывается от завершения подключения:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
f1:6f:ea:87:bb:1b:df:cd:e3:45:24:60:d3:25:b1:0a.
Please contact your system administrator.
Add correct host key in /home/sam/.ssh/known_hosts to get rid of this message.
Offending key in /home/sam/.ssh/known_hosts:1
RSA host key for plum has changed and you have requested strict checking.
Host key verification failed.
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    ВНИМАНИЕ: ИДЕНТИФИКАЦИЯ УДАЛЕННОГО ХОСТА ИЗМЕНИЛАСЬ!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
ВОЗМОЖНО, ЭТО ЧЬИ-ТО ПРОДЕЛКИ!
Кто-то именно сейчас может перехватывать вашу информацию (man-in-the-middle attack)!
Возможно также, что только что был изменен RSA хост-ключ.
Отпечаток RSA-ключа, отправленный удаленным хостом, имеет следующий вид
f1:6f:ea:87:bb:1b:df:cd:e3:45:24:60:d3:25:b1:0a.
Пожалуйста, свяжитесь со своим системным администратором.
Добавьте правильный ключ в файл /home/sam/.ssh/known_hosts, чтобы избавиться
от этого сообщения.
Неверный ключ находится в /home/sam/.ssh/known_hosts:1
RSA хост-ключ для plum был изменен, и от вас требуется тщательная проверка.
Верификация хост-ключа не состоялась.
```

Получение такого сообщения может быть признаком состоявшейся посреднической атаки (man-in-the-middle attack). Но, скорее всего, что-то изменилось на удаленной системе, что заставило ее предоставить новый отпечаток. Нужно оценить ситуацию с помощью администратора удаленной системы. Если все в порядке, удалите

неверный ключ из указанного файла (в предыдущем примере строка, которую нужно удалить, указана в третьей строчке с конца) и попытайтесь снова установить соединение. Для удаления ненужной записи можно воспользоваться утилитой ssh-keygen с ключом -R, за которым следует имя хоста. Следуя этому сценарию, вы будете опять проходить первоначальную аутентификацию, как только OpenSSH проверит, что вы подключились к правильной системе. Нужно будет сделать все то же самое, что и при начальном подключении к удаленному хосту.

Ключи авторизации: автоматический вход в систему

OpenSSH можно настроить так, чтобы не нужно было вводить пароль при каждом подключении к серверу (к удаленной системе). Для настройки этого свойства нужно сгенерировать персональный идентификационный ключ на клиенте (на локальной системе), поместить открытую часть ключа на сервер и хранить закрытую часть ключа на клиенте. При подключении к серверу он выдает запрос на основе открытой части ключа. Затем закрытая часть ключа должна правильно ответить на этот запрос. Если клиент предоставляет допустимый ответ, сервер позволяет войти в систему.

Первым шагом на пути настройки автоматического входа в систему должна стать генерация персональных ключей аутентификации. Сначала нужно проверить наличие этих ключей на локальной системе (клиенте), посмотрев, нет ли в каталоге ~/.ssh либо файлов id_dsa и id_dsa.pub, либо файлов id_rsa и id_rsa.pub. Если одна из этих пар там имеется, следующий шаг можно пропустить (то есть не создавать новый ключ).

Утилита ssh-keygen создает на клиенте открытую и закрытую части RSA-ключа. Произвольная картинка ключа — The key's randomart image — это визуальная репрезентация открытого ключа; она создается с расчетом на то, что ее не трудно будет вспомнить. Вывод произвольной картинки клиентом управляется путем объявления свойства VisualHostKey в файле ssh_config.

ssh-keygen

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
  Генерация пары из открытой и закрытой частей rsa-ключа
Enter file in which to save the key (/home/sam/.ssh/id_rsa):ВВОД
  Введите файл, в котором нужно сохранить ключ (/home/sam/.ssh/id_rsa):
Enter passphrase (empty for no passphrase): ВВОД
  Введите идентификационную фразу (если она не нужна, ничего не вводите):
Enter same passphrase again: ВВОД
  Введите ту же самую идентификационную фразу еще раз:
Your identification has been saved in /home/sam/.ssh/id_rsa.
  Ваша идентификация сохранена в файле /home/sam/.ssh/id_rsa.
Your public key has been saved in /home/sam/.ssh/id_rsa.pub.
  Открытая часть вашего ключа сохранена в файле /home/sam/.ssh/id_rsa.pub.
The key fingerprint is:
  Отпечаток ключа имеет следующий вид:
f2:eb:c8:fe:ed:fd:32:98:e8:24:5a:76:1d:0e:fd:1d sam@peach
The key's randomart image is:
  Произвольная картинка ключа имеет следующий вид:
+--[ RSA 2048]----+
|               oE|
|     o .       . o|
|    . o +       ..|
|   .   + o      o |
...
```

Для генерации DSA-ключей нужно вместо rsa указать dsa. В данном примере пользователь нажимал ВВОД в ответ на каждый запрос. У вас есть возможность указать идентификационную фразу (ее оптимальная длина — 10–30 символов) для шифрования закрытой части ключа. Способов восстановления утраченной идентификационной фразы не существует. Дополнительные сведения об идентификационной фразе даны в следующей врезке «Безопасность».

ЧТО НУЖНО ЗНАТЬ ПРИ ШИФРОВКЕ ПЕРСОНАЛНОГО КЛЮЧА

Закрытая часть ключа хранится в файле, который можете прочитать только вы. Если злоумышленник взломает вашу учетную запись, с помощью которой можно воспользоваться утилитой sudo для получения root-прав, или взломает корневую учетную запись (root) на локальной системе, то затем он сможет получить доступ к вашей учетной записи на удаленной системе, поскольку сможет прочитать закрытую часть вашего персонального ключа.

Шифровка закрытой части вашего персонального ключа защитит ключ и тем самым ограничит доступ к удаленной системе, если кто-нибудь взломает вашу учетную запись на локальной системе. Но при шифровке своего персонального ключа нужно предоставить идентификационную фразу, которая используется для расшифровки ключа при каждом его использовании, сводя на нет все преимущества от отсутствия необходимости набора пароля при каждом входе на удаленную систему. К тому же большинство легко запоминаемых идентификационных фраз могут быть довольно быстро подобраны на мощном компьютере.

Лучше всего хранить закрытый ключ на съемном носителе, например на USB флэш-накопителе, и использовать свой каталог ~/.ssh как точку монтировки файловой системы, хранящейся на этом накопителе. А шифровка этих ключей может понадобиться при потере флэш-накопителя.

Утилита ssh-keygen генерирует два ключа: закрытый ключ или идентификацию в файле ~/.ssh/id_rsa и открытый ключ в файле ~/.ssh/id_rsa.pub. Никто кроме владельца не сможет ничего записать ни в один из этих файлов, и только владелец может что-нибудь прочитать из файла с закрытым ключом.

authorized_keys

Чтобы получить возможность входа в систему или копирования файлов на другую систему или из нее без предоставления пароля, сначала на сервере (удаленной системе) нужно создать каталог ~/.ssh с набором прав доступа, установленным в 700. Затем скопировать файл ~/.ssh/id_rsa.pub с клиента (локальной системы) в файл по имени ~/.ssh/authorized_keys на сервере (удаленной системе). Установить для него права доступа 600, чтобы никто кроме владельца не мог читать файл и вести в него запись. Теперь при запуске ssh или scp для получения доступа к серверу предоставлять пароль не придется. Чтобы повысить степень безопасности сервера, можно выключить парольную аутентификацию, установив для свойства PasswordAuthentication в файле /etc/ssh/sshd_config значение «no» (удалить комментирующий символ # из начала строки PasswordAuthentication и изменить «yes» на «no»).

Примеры

В первом примере пользователь Max использует ssh для вывода списка файлов в своем личном каталоге на системе kudos:

```
$ ssh kudos ls
max@kudos's password:
Work
code
graphs
reports
```

Затем вывод из предыдущей команды перенаправляется в файл kudos_ls. Поскольку символ перенаправления (>) не отключен, он интерпретируется локальной оболочкой, и файл kudos_ls создается на локальной системе.

```
$ ssh kudos ls > kudos_ls
max@kudos's password:
$ cat kudos_ls
Work
code
graphs
reports
```

В следующем примере в кавычки берется вся команда, запускаемая на удаленной системе. В результате локальная оболочка не интерпретирует символ перенаправления (>), а передает его удаленной оболочке. Файл kudos.ls создается на удаленной системе (kudos), что и показывает команда ls, запущенная на kudos:

```
$ ssh kudos "ls > kudos.ls"
max@kudos's password:
$ ssh kudos ls
max@kudos's password:
Work
code
graphs
kudos.ls
reports
```

Следующая команда не цитирует символ канала (|). В результате этого канал интерпретируется локальной оболочкой, которая отправляет вывод удаленной команды ls на стандартный ввод команды less на локальной системе:

```
$ ssh kudos ls | less
```

Затем утилита ssh выполняет ряд команд на удаленной системе, которые соединены друг с другом каналами. Команды заключены в одинарные кавычки, поэтому локальная оболочка не интерпретирует символы канала, и все команды запускаются на удаленной системе.

```
$ ssh kudos 'ps -ef | grep nmbd | grep -v grep | cut -c10-15 |xargs kill -1'
```

Вывод команды ps проводится по каналу через команду grep, которая передает все строки, содержащие последовательность символов nmbd, другой вызванной команде grep. Вторая команда grep передает все строки, не содержащие последовательности символов grep, команде cut (см. описание соответствующей утилиты). Утилита cut извлекает ID-номера процессов и передает их утилите xargs (см. описание этой утилиты), которая прекращает работу перечисляемых процессов, вызывая утилиту kill для выдачи сигнала HUP (kill -1).

В следующем примере утилита ssh без ***командной_строки*** осуществляет вход на удаленную систему. Здесь пользователь Max воспользовался аргументом watson@kudos для входа в систему kudos под именем watson:

```
$ ssh watson@kudos
watson@kudos's password:
Last login: Sat Sep 17 06:51:59 from bravo
```

```
$ hostname
kudos
$ exit
```

Теперь Max решил изменить пароль для своего входа под именем watson на систему kudos:

```
$ ssh watson@kudos passwd
watson@kudos's password:
(current) UNIX password: por
```

Max прекратил ввод, как только увидел, что команда passwd (запущенная на kudos) вывела его пароль: он понял, что что-то тут не так. Чтобы команда passwd работала, ее нужно запустить вместе с tty (терминалом), чтобы она могла выключить вывод символов (stty -echo), тогда пароли не будут выводиться по мере их ввода пользователем. Эту проблему решает использование ключа -t, который связывает с процессом, запустившим команду passwd на удаленной машине, псевдо-tty:

```
$ ssh -t watson@kudos passwd
watson@kudos's password:                    пароль для watson@kudos:
Changing password for watson                Изменение пароля для watson
(current) UNIX password:                    UNIX-пароль (текущий):
New UNIX password:                          Новый UNIX-пароль:
Retype new UNIX password:                   Еще раз наберите новый UNIX-пароль:
passwd: all authentication tokens updated successfully
   passwd: все элементы аутентификации обновлены успешно
Connection to kudos closed.                 Подключение к kudos закрыто
$
```

Ключ -t применяется также при запуске программы, которая использует основанный на обычных символах псевдографический интерфейс.

В следующем примере утилита tar (см. соответствующее описание) используется для создания архивного файла с содержимым иерархии рабочего каталога. Ключ f – заставляет утилиту tar отправлять ее вывод на стандартный вывод. Канал переправляет вывод утилиты tar, запущенной на локальной системе, через ssh утилите dd (см. соответствующее описание), запущенной на удаленной системе.

```
$ cat buwd
#! /bin/bash
# резервное копирование рабочего каталога в личный
# каталог пользователя на удаленной системе, указанный
# с помощью переменной $machine

# удаленная система:
machine=speedy

dir=$(basename $(pwd))
filename=$$.$dir.tar

echo Резервное копирование $(pwd) на ваш личный каталог на $machine
tar -cf - . | ssh $machine "dd obs=256k of=$filename"
echo Готово. Имя файла на $machine - $filename
```

stat

Вывод информации о файлах.

stat [ключи] [список_файлов]

Утилита stat выводит информацию о файлах.

Аргументы

Список_файлов указывает имя одного или нескольких файлов, о которых утилита stat выводит информацию. Без ***списка_файлов*** stat выводит информацию о стандартном вводе.

Ключи

Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

Без ключей утилита stat выводит всю доступную информацию о каждом обрабатываемом файле.

`--format=`*fmt*
`-c` *fmt* — Форматирование вывода с использованием аргумента *fmt*. Дополнительные сведения даны на man-странице, посвященной утилите stat. (Linux)

`-F` — (file type — тип файла). Вывод слэша (/) после имени каждого каталога, звездочки (*) после имени каждого исполняемого файла, знака at (@) после имени символьной ссылки, знака равенства (=) после имени сокета, и знака канала (|) после имени FIFO. (OS X)

`-f` — (filesystem — файловая система). Вывод информации о файловой системе, а не о файле. (Linux)

`-f` ***формат*** — Форматирование вывода с помощью аргумента *формат*. Строка ***формат*** аналогична той, которая используется в команде printf. Дополнительные сведения даны на man-странице, посвященной утилите stat. (OS X)

`--dereference`
`-L` — Для каждого файла, являющегося символьной ссылкой, вывод информации о файле, на который указывает ссылка, а не о самой символьной ссылке. При использовании этого ключа файлы, не являющиеся символьными ссылками, обрабатываются обычным образом.

`-l` — (long — длинный формат). Использование такого же формата, как и у команды `ls -l` (см. главу 4, раздел «ls –l: Отображение прав»). (OS X)

`-q` — (quiet — молчание). Подавление вывода сообщений об ошибках. (OS X)

`-s` — (shell — оболочка). Вывод информации в формате, который может быть использован для инициализации переменных оболочки. См. подраздел «Примеры». (OS X)

`-x` — (Linux). Вывод более полного формата, совместимого с версией stat, используемой в Linux. (OS X)

Примеры

Версии утилиты stat, используемые в Linux и в Mac OS X, выдают разную информацию. В примерах показана информация, выводимая Linux-версией.

В первом примере выводится информация о файле /bin/bash:

```
$ stat /bin/bash
  File: '/bin/bash'
  Size: 813912          Blocks: 1600       IO Block: 4096   regular file
Device: 812h/2066d      Inode: 146674      Links: 1
Access: (0755/-rwxr-xr-x)  Uid: ( 1003/     sam)   Gid: ( 1002/    pubs)
Access: 2009-05-28 11:36:01.000000000 -0700
Modify: 2008-05-12 11:36:49.000000000 -0700
Change: 2009-05-21 11:34:41.000000000 -0700
```

В следующем примере выводится информация о корневой файловой системе:

```
$ stat -f /
  File: "/"
    ID: 586ac700c664ac38 Namelen: 255     Type: ext2/ext3
Block size: 4096       Fundamental block size: 4096
Blocks: Total: 960602     Free: 703655     Available: 654858
Inodes: Total: 488640     Free: 467723
```

strings

Вывод строк, состоящих из символов, которые можно вывести на печать.

strings [ключи] список_файлов

Утилита strings выводит строки из выводимых на печать символов из объектных и других нетекстовых файлов.

Аргументы

Список_файлов — это список файлов, обрабатываемых утилитой strings.

Ключи

Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--all -a` Обработка всех файлов целиком. Без этого ключа утилита strings обрабатывает только инициализирующую и загрузочную части объектного файла.

`--print-file-name -f` Установка в начале каждой строки имени файла, откуда эта строка была взята. (Linux)

`--bytes=`*min* `-`*min* Вывод строк символов, имеющих длину хотя бы в *min* символов (по умолчанию эта длина равна 4).

Комментарии

Утилита strings может помочь определить суть содержимого нетекстовых файлов. Одним из известных применений утилиты strings является определение владельца файлов в каталоге lost+found.

Примеры

В следующем примере выводятся строки из четырех и более выводимых на печать символов в исполняемом файле утилиты man. Если не знать, чем был этот файл, эти строки могут помочь определить, что это был исполняемый файл утилиты man.

```
$ strings /usr/bin/man
…
--Man-- next: %s [ view (return) | skip (Ctrl-D) | quit (Ctrl-C) ]
format: %d, save_cat: %d, found: %d
cannot write to %s in catman mode
creating temporary cat for %s
can't write to temporary cat for %s
can't create temporary cat for %s
cat-saver exited with status %d
found ultimate source file %s
…
```

stty

Вывод или установка параметров терминала.

stty [ключи] [аргументы]

Без аргументов утилита stty выводит параметры, которые влияют на работу терминала или эмулятора терминала. Список некоторых из этих параметров и разъяснения по каждому из них даны в подразделе «Комментарии». ***Аргументы*** устанавливают или изменяют параметры.

Ключи

При работе под управлением Linux утилита stty допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--all -a` Вывод сведений о всех параметрах. Этот ключ не допускает использования аргументов.

`--file=`*устройство* `-F` *устройство* Воздействие на устройство. Без этого ключа stty воздействует на устройство, подключенное к стандартному вводу. Характеристики устройства можно изменить только являясь владельцем его файла или работая с root-правами. (Linux)

`-f` *устройство* Воздействие на устройство. Выполнение той же функции, что и у ключа `-F`.(OS X)

`--save -g` Генерирование отчета о текущих установках в формате, который можно затем использовать в качестве аргументов для другой команды stty. Этот ключ не допускает использования аргументов.

Аргументы

Аргументы для утилиты stty указывают, какой из параметров терминала должна изменить утилита stty. Для включения каждого из параметров, перед которыми ставится дополнительный одинарный дефис (показанный в следующем списке как `[-]`), нужно указать параметр без дефиса. Чтобы отключить параметр, используется дефис. Кроме особо оговоренных случаев, параметры в данном разделе описываются в их включенных состояниях.

Специальные клавиши и характеристики

`columns` *n* Установка ширины строки в *n* столбцов.

`ek` (erase kill — стирание и удаление). Установка клавиш стирания и удаления строк на их значения по умолчанию. Многие системы используют по умолчанию клавиши DELETE и CONTROL+U соответственно.

`erase` *x* Установка *x* в качестве клавиши стирания. Для указания управляющего символа нужно перед *x*, нажать CONTROL+V (например, использовать комбинацию CONTROL+V CONTROL+H для отображения CONTROL+H) или воспользоваться системой записи ^h, где ^ — это символ вставки (на большинстве клавиатур — SHIFT+6).

`intr` *x* Установка *x* в качестве клавиши прерывания. Правила установки см. в описании аргумента `erase` *x*.

`kill` *x* Установка *x* в качестве клавиши удаления строки. Правила установки см. в описании аргумента `erase` *x*.

`rows` *n* Установка количества экранных строк на *n*.

`sane` Установка параметров терминала на те значения, которые обычно являются приемлемыми. Аргумент sane применяется в том случае, когда были изменены сразу несколько параметров stty, затруднив тем самым использование терминала для запуска stty с целью правильной настройки. Если работа аргумента sane не видна, следует попробовать ввести следующие символы:
`CONTROL+J stty sane CONTROL+J`

`susp` *x* (suspend — приостановка). Установка *x* в качестве клавиши приостановки (остановки работы терминала). Правила установки см. в описании аргумента `erase` *x*.

`werase` *x* (word erase — стирание слова). Установка *x* в качестве клавиши стирания слова. Правила установки см. в описании аргумента `erase` *x*.

Режимы передачи данных

`[-]cooked` См. `raw`. (Linux)

`cooked` См. `sane`. (OS X)

`[-]cstopb` (stop bits — стоповые биты). Выбор двух стоповых битов (`-cstopb` указывает использование одного стопового бита).

`[-]parenb` (parity enable — включение бита четности). Включение бита четности для ввода и вывода. При указании `-parenb` система не использует или не ожидает бита четности при обмене данными с терминалом.

`[-]parodd` (parity odd — проверка на нечетность). Выбор проверки на нечетность (`-parodd` указывает на выбор проверки на четность).

`[-]raw` Нормальное состояние `-raw`. Когда система читает ввод в необработанном виде, она не интерпретирует следующие специальные символы: стирания (обычно DELETE), удаления строки (обычно CONTROL+U), выполнения прерывания (CONTROL+C) и конца файла — EOF (CONTROL+D). Кроме этого, система не использует биты четности. Отдавая должное свойственному создателям Linux юмору, при работе под Linux можно также указать `-raw` как `cooked`.

Обработка символов

`[-]echo` Выведение символов по мере их набора (полнодуплексная операция). Если терминал полудуплексный и отображает два символа вместо одного подлежащего выводу на экран, параметр `echo` нужно отключить (`-echo`). Аргумент `-echo` нужно использовать при вводе пользователем паролей.

`[-]echoe` (echo erase). Обычно установлен режим `echoe`, который заставляет ядро выводить при использовании клавиши стирания для удаления символа символьную последовательность ЗАБОЙ ПРОБЕЛ ЗАБОЙ (BACKSPACE SPACE BACKSPACE). В результате курсор по мере удаления символов возвращается назад по строке.

`[-]echoke` (echo kill erase). Обычно установлен режим `echoke`. Когда при установленном режиме используется символ удаления строки, в текущей строке удаляются все символы, возвращая курсор в позицию приглашения на ввод. Когда этот режим отключен, нажатие клавиши удаления приводит к перемещению курсора к началу следующей строки.

`[-]echoprt` (echo print). Обычно установлен режим `-echoprt`, заставляющий символы исчезать по мере их стирания. При установке режима `echoprt` стираемые символы отображаются между обратным (`\`) и прямым слэшем (`/`). Например, если набрать слово `sort`, а затем стереть его, четыре раза нажав ЗАБОЙ, то при включенном режиме Linux выведет `sort\tros/`. Когда для удаления всей строки используется клавиша удаления, включенный режим `echoprt` заставляет всю строку выводиться как будто все символы стерты клавишей ЗАБОЙ до начала строки.

`[-]lcase` Для терминалов, работающих только с символами верхнего регистра, перевод всех символов верхнего регистра в символы нижнего регистра по мере их ввода (также работает и аргумент `[-]LCASE`). (Linux)

[-]nl Допуск в качестве символа, завершающего строку, разделителя строк (NEWLINE). При включении режима -nl система воспринимает символ ВВОД, поступающий с терминала, как разделитель строк, но вместо разделителя строк (NEWLINE) отправляет на терминал ВВОД (RETURN), за которым следует разделитель строк (NEWLINE).

[-]tabs Передача на терминал каждого символа табуляции в его натуральном виде. При выключенном свойстве (-tabs) ядро превращает каждый символ табуляции в соответствующее количество пробелов и передает их на терминал (также работает и аргумент [-]tab3).

Параметры управления заданиями

[-]tostop Остановка фоновых заданий, если они пытаются отправить вывод на терминал (-tostop позволяет фоновым заданиям отправлять вывод на терминал).

Примечания

Название stty является аббревиатурой set teletypewriter (установка параметров телетайпа), или set tty (см. соответствующее описание), первого терминала, с которым работала система UNIX. Сегодня stty обычно ассоциируется с настройкой терминала (set terminal).

При интерактивном использовании оболочек они сохраняют определенный контроль над стандартным вводом. Вследствие этого, многие выбранные с помощью stty режимы работы не имеют никакого эффекта. Например, команда stty -echo под управлением tcsh не срабатывает:

```
tcsh $ stty -echo
tcsh $ date
Thu May 28 16:53:01 PDT 2009
```

Если команда stty -echo при использовании оболочки bash в интерактивном режиме работает, то команда stty -echoe — нет. Но все эти настройки по-прежнему можно использовать для изменения характера работы сценариев оболочки и других утилит.

```
$ cat testit
#!/bin/bash
stty -echo
echo -n "Введите значение: "
read a
echo
echo "Вы ввели: $a"
stty echo

$ ./testit
Введите значение:
Вы ввели: 77
```

В предыдущем примере пользовательский ответ на приглашение Введите значение: не выводится. Значение сохраняется в переменной a, и выводится инструкцией echo "Вы ввели: $a".

Примеры

В первом примере показывается, что утилита stty без аргументов выводит несколько параметров работы терминала. (Ваша система может вывести больше параметров или несколько другие параметры.) Символ, следующий за erase =, является обозначением клавиши стирания. Знак вставки ^, предшествующий буквенному символу, отображает клавишу CONTROL. В данном примере клавиша стирания настроена на комбинацию CONTROL+H. Если stty не показывает настройку клавиши стирания, значит, она настроена на свое значение по умолчанию — DELETE. Если не показывается настройка клавиши удаления, значит, она настроена на свое значение по умолчанию — ^U.

```
$ stty
speed 38400 baud; line = 0;
erase = ^H;
```

Затем использование аргумента ek возвращает настройку клавиш стирания и удаления строки к значениям по умолчанию:

```
$ stty ek
```

Следующее отображение настроек позволяет проверить изменения. Утилита stty не показывает настройки ни клавиши стирания, ни клавиши удаления строки, что свидетельствует о том, что обе они имеют настройки по умолчанию:

```
$ stty
speed 38400 baud; line = 0;
```

В следующем примере клавиша стирания настраивается на CONTROL+H. Использование комбинации CONTROL+V приводит к цитированию комбинации CONTROL+H, поэтому оболочка ее не интерпретирует и передает утилите stty без изменений:

```
$ stty erase CONTROL+VCONTROL+H
$ stty
speed 38400 baud; line = 0;
erase = ^H;
```

Затем утилита stty настраивает клавишу удаления строки на комбинацию CONTROL+X. На этот раз пользователь вводит знак вставки (^), за которым следует символ x, что является представлением комбинации CONTROL+X. Букву можно использовать как в нижнем, так и в верхнем регистре.

```
$ stty kill ^X
$ stty
speed 38400 baud; line = 0;
erase = ^H; kill = ^X;
```

А теперь утилита stty изменяет клавишу прерывания на комбинацию CONTROL+C:

```
$ stty intr CONTROL+VCONTROL+C
```

В следующем примере утилита stty отключает отправку символов табуляции, вместо них на терминал отправляется соответствующее количество пробелов. Если терминал не производит автоматическое расширение символов табуляции, нужно воспользоваться следующей командой:

```
$ stty -tabs
```

Если вход в систему осуществляется под управлением Linux и вся выводимая на терминал информация появляется в буквах верхнего регистра, нужно ввести следую-

щую команду, а затем проверить работу клавиши CAPS LOCK. Если установлен верхний регистр, его нужно выключить.

```
$ STTY -LCASE
```

Если используется очень старый терминал, неспособный выводить символы в нижнем регистре, нужно включить параметр `lcase`.

Хотя значение по умолчанию CONTROL+Z для клавиши приостановки обычно никем не изменяется, это тоже можно сделать. Чтобы изменить значение для этой клавиши на CONTROL+T, нужно ввести следующую команду:

```
$ stty susp ^T
```

sysctl (OS X)

Вывод значений и изменение переменных ядра.

sysctl [ключ] [список_переменных]
sysctl [ключ] –w [переменная=значение ...]

Утилита sysctl выводит и изменяет значения переменных ядра, включая параметры его настройки.

Аргументы

Список_переменных — это список переменных ядра, чьи значения выводит утилита sysctl. Во втором формате каждое ***значение*** присваивается переменной по имени ***переменная***.

Ключи

`-a` (all — все). Вывод значений всех переменных ядра.

`-b` (binary — двоичный). Вывод значений переменных ядра в виде двоичных данных без их разрыва разделителями строк.

`-n` (no label — без метки). Вывод значений переменных без меток.

Комментарии

Утилита sysctl предоставляет доступ к ряду переменных ядра, включая максимальное количество процессов, одновременно запущенных ядром, имя, используемое для файлов ядра, и уровень безопасности системы. Значение некоторых переменных не может быть изменено или может изменяться конкретными способами. Например, вы никогда не сможете понизить уровень безопасности.

Примеры

Утилита sysctl обычно используется для настройки ядра. Лимит процессов может быть выведен любым пользователем, но изменить его может только пользователь, работающий с root-правами. В следующем примере показываются действия пользователя, изменяющего максимальное количество процессов, относящихся к одному UID:

```
$ sysctl -a | grep proc
kern.maxproc = 532
kern.maxfilesperproc = 10240
kern.maxprocperuid = 266
kern.aioprocmax = 16
kern.proc_low_pri_io = 0
…
# sysctl -w kern.maxprocperuid=200
kern.maxprocperuid: 266 -> 200
```

tail

Вывод последней (заключительной) части файла.

tail [ключи] [список_файлов]

Утилита tail выводит последнюю часть, или «хвост» файла.

Аргументы

Список_файлов — это список путевых имен файлов, чьи окончания выводит утилита tail. Если указано более одного файла, tail выводит имя каждого файла перед выводом его строк. Если не указать аргумент или если при работе под управлением Linux указать вместо имени файла дефис (-), утилита tail осуществляет чтение со стандартного ввода.

Ключи

При работе под управлением Linux утилита stty допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме особо оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`-b` ***[+]n*** Вычисление выводимой части файла не в строках, а в 512-байтовых блоках. В качестве аргумента *n* используется целое число, указывающее количество блоков. Таким образом, команда `tail -b 5` выводит последние пять блоков файла. Примечание по использованию знака плюс (+) дано в описании следующего ключа. (OS X)

`--bytes=`***[+]n[u]*** `-c` ***[+]n[u]*** Вычисление выводимой части файла не в строках, а в байтах (символах). В качестве аргумента *n* используется целое число, указывающее количество байтов. Таким образом, команда `tail -c 5` выводит последние пять байтов файла. Аргумент *u* используется только при работе под Linux и является необязательным мультипликативным суффиксом из числа тех, которые рассмотрены в разделе «Стандартные мультипликативные суффиксы», за исключением того, что утилита tail использует для килобайтов (1024-байтных блоков) букву k в нижнем регистре. Если включен мультипликативный суффикс, tail ведет вычисления не в байтах, а в этой единице измерения.

Если перед *n* поставить знак плюса (+), утилита tail будет вести вычисления с начала файла, а не с его конца. Она продолжает выводить символы до конца файла, но теперь вычисление будет вестись с начала файла. Таким образом, команда `tail -c +5` заставляет утилиту tail вывести содержимое файла с пятого символа и до конца файла.

`--follow -f` После копирования последней строки файла утилита tail входит в бесконечный цикл, ожидая и копируя дополнительные строки из файла по мере его пополнения. Если в ***списке_файлов*** с этим ключом указано несколько файлов, утилита tail включает новый заголовок при каждом отображении вывода из другого файла, чтобы дать возможность узнать, какой из файлов был дополнен. Этот ключ применяется для отслеживания хода процесса, запущенного в фоновом режиме и отправляющего свой вывод в файл. Утилита tail продолжает свое ожидание бесконечно, поэтому для завершения ее работы нужно воспользоваться клавишей прерывания. См. также описание ключа `-s`.

`--lines=`*[+]n[u]* `-n` *[+]n[u]* Вычисление выводимой части файла в строках (используется по умолчанию). В качестве аргумента *n* используется целое число, указывающее количество строк. При работе под управлением Linux аргумент *u* является необязательной единицей измерения, порядок его использования объяснен в описании ключа `-c` (`--bytes`). Для указания количества строк можно обойтись и без этого ключа, воспользовавшись незадокументированным аргументом ±*n*.

Если перед *n* поставить знак плюса (+), утилита tail будет вести вычисления с начала файла, а не с его конца. Она продолжает выводить строки до конца файла, но теперь вычисление будет вестись с начала файла. Таким образом, команда `tail -n +5` заставляет утилиту tail вывести содержимое файла с пятой строки и до конца файла.

`--quiet -q` Подавление вывода заголовочной информации при указании в ***списке_файлов*** сразу нескольких файлов.(Linux)

`--sleep-interval=`*n* `-s` *n* При использовании с ключом `-f` заставляет утилиту tail приостанавливать работу на *n* секунд, перед тем как она начнет проверять наличие дополнительного вывода. (Linux)

Примечания

По умолчанию утилита tail выводит последние десять строк своего вывода.

Примеры

В примерах используется файл по имени eleven:

```
$ cat eleven
строка один
строка два
строка три
строка четыре
строка пять
строка шесть
строка семь
строка восемь
```

```
строка девять
строка десять
строка одиннадцать
```

Сначала утилита tail выводит последние десять строк файла eleven (ключи не используются):

```
$ tail eleven
строка два
строка три
строка четыре
строка пять
строка шесть
строка семь
строка восемь
строка девять
строка десять
строка одиннадцать
```

Затем она выводит последние три строки (`-n 3` или `--lines 3`) файла:

```
$ tail -n 3 eleven
строка девять
строка десять
строка одиннадцать
```

В следующем примере содержимое файла выводится начиная с восьмой строки (+8):

```
$ tail -n +8 eleven
строка восемь
строка девять
строка десять
строка одиннадцать
```

В следующем примере выводятся последние шесть символов файла (`-c 6` или `--bytes 6`). Показываются только пять символов (дцать), шестым является разделитель строк (NEWLINE).

```
$ tail -c 6 eleven
дцать
```

В последнем примере показывается действие ключа `-f`. Здесь утилита tail позволяет отслеживать вывод команды make, который был отправлен в файл accounts.out:

```
$ make accounts > accounts.out &
$ tail -f accounts.out
        cc -c trans.c
        cc -c reports.c
...
CONTROL+C
$
```

В предыдущем примере запуск утилиты tail с ключом `-f` приводит к отображению такой же информации, которая появляется при запуске команды make в фоновом режиме без перенаправления ее вывода в файл. Тем не менее использование утилиты tail предлагает ряд преимуществ. Во-первых, вывод из команды make сохраняется в файле. (Вывод не будет сохранен, если его не перенаправить.) Во-вторых, если будет при-

нято решение заняться на время работы make чем-нибудь другим, можно прекратить работу утилиты tail и освободить экран для себя, пока make будет продолжать работу в фоновом режиме. При запуске продолжительных заданий, таких как компиляция больших программ, можно использовать утилиту tail с ключом `-f` для периодической проверки состояния дел.

tee

Копирование стандартного ввода на стандартный вывод, а также в один или несколько файлов.

tee [ключи] список_файлов

Утилита tee копирует стандартный ввод на стандартный вывод, а также в один или несколько файлов.

Аргументы

Список_файлов — это список путевых имен файлов, получающих вывод утилиты tee. Если файл, указанный в ***списке_файлов*** не существует, утилита tee его создает.

Ключи

Ключи, перед которыми ставится двойной дефис (`--`), работают только под управлением Linux. Однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

Без ключей утилита tee переписывает файлы вывода, если они существуют, и реагирует на прерывания.

`--append -a` Добавление вывода к существующим файлам вместо их переписывания.

`-i` (OS X) Принуждение утилиты tee не реагировать на прерывание `SIGINT`.

`--ignore-interrupts` `-i` Принуждение утилиты tee не реагировать на прерывания. (Linux)

Примеры

В следующем примере канал отправляет вывод из утилиты make утилите tee, которая копирует эту информацию на стандартный вывод и в файл accounts.out. Копия, направляемая на стандартный вывод, появляется на экране. Утилита cat выводит копию, отправленную в файл.

```
$ make accounts | tee accounts.out
        cc -c trans.c
        cc -c reports.c
...
```

```
$ cat accounts.out
        cc -c trans.c
        cc -c reports.c
...
```

Похожий пример, в котором вместо утилиты tee используется утилита tail с ключом `-f`, показан в описании утилиты tail.

telnet

Подключение к удаленной системе по сети.

telnet [ключи] [удаленная_система]

Утилита telnet выполняет протокол TELNET для подключения к удаленной системе по сети.

УТИЛИТА TELNET НЕ ОБЕСПЕЧИВАЕТ ДОЛЖНУЮ СТЕПЕНЬ БЕЗОПАСНОСТИ

Утилита telnet не обеспечивает безопасность подключения. Она отправляет ваше имя пользователя и пароль по сети открытым текстом, что не отвечает требованиям безопасности. По возможности следует использовать утилиту ssh (см. описание этой утилиты).

Аргументы

Удаленная_система — это имя или IP-адрес удаленной системы, к которой подключается утилита telnet. Если ***удаленная_система*** не указана, telnet работает в интерактивном режиме и выводит приглашение на ввод одной из команд, описанных в данном разделе.

Ключи

`-a` Инициирование автоматического входа в систему (при работе под управлением Mac OS X, это делается по умолчанию).

`-e` *c* (escape). Изменение символа выхода с CONTROL+] на символ *c*.

`-K` Предотвращение автоматического входа в систему. Этот ключ доступен при работе под управлением Mac OS X и некоторых дистрибутивов Linux (при работе под управлением Linux это делается по умолчанию).

`-l` *имя_пользователя* Попытка автоматического входа на удаленную систему под именем ***имя_пользователя***. Если удаленная система способна обработать автоматический вход в систему с использованием telnet, то она выводит приглашение на ввод пароля.

Комментарии

После того как telnet подключится к удаленной системе, эту утилиту можно перевести в командный режим, набрав символ выхода (обычно это CONTROL+]). Удаленная система обычно сообщает о том, что распознала символ выхода. Чтобы выйти из командного режима, нужно, находясь в командной строке, нажать только клавишу ВВОД.

В командном режиме утилита **telnet** выводит приглашение `telnet>`. После этого можно использовать следующие команды:

`?` (help — справка). Вывод перечня команд, распознаваемых утилитой **telnet** на локальной системе.

`close` Закрытие подключения к удаленной системе. Если при запуске **telnet** в командной строке было указано имя системы, `close` имеет такое же действие, как и команда `quit`: программа **telnet** завершает работу, и оболочка выводит приглашение. Если вместо указания в командной строке удаленной системы использовалась команда `open`, то команда `close` возвращает **telnet** в командный режим.

`logout` Осуществление выхода из удаленной системы; то же самое, что и `close`.

`open` *удаленный_компьютер* Если удаленная система в командной строке не указана или если попытка подключения к системе не удалась, можно указать имя удаленной системы в интерактивном режиме с помощью команды `open`.

`quit` Завершение сеанса работы **telnet**.

`z` Приостановка сеанса **telnet**. При такой приостановке происходит возврат к оболочке входа в локальную систему. Для продолжения приостановленного сеанса **telnet** следует вызвать в приглашении оболочки команду **fg**.

Примечания

При работе под управлением Linux утилита **telnet** не пытается войти в систему в автоматическом режиме. А при работе под управлением OS X эта утилита пытается осуществить автоматический вход. Когда **telnet** пытается войти в систему в автоматическом режиме, он использует ваше имя пользователя на локальной системе, если не будет указано другое имя с помощью ключа `-l`.

Протокол TELNET поддерживается многими компьютерами, включая системы, работающие не под управлением Linux или OS X. Утилита **telnet** является пользовательским интерфейсом к этому протоколу для систем, работающих под Linux или под OS X, который позволяет подключаться ко многим различным типам систем. Хотя обычно **telnet** используется для входа в систему, удаленный компьютер может предложить для работы через **telnet**, например, доступ к специализированным базам данных.

Примеры

В следующем примере пользователь подключается к удаленной системе по имени **bravo**. После запуска нескольких команд пользователь выходит из командного режима и использует команду `z` для приостановки сеанса **telnet**, чтобы получить возможность запуска нескольких команд на локальной системе. Для возобновления использования **telnet** пользователь дает оболочке команду `fg`. Команда `logout` на удаленной системе завершает сеанс **telnet**, и локальная оболочка выводит приглашение.

```
kudos% telnet bravo
Trying 192.168.0.55 …
Connected to bravo.                    Произошло подключение к bravo
Escape character is '^]'.              Символ выхода – '^]'.
```

```
login: watson
Password:
Last login: Wed Jul 31 10:37:16 from kudos      Последнее подключение: … с kudos
bravo $
…
bravo $CONTROL+]
telnet> z

[1]+ Stopped  telnet bravo                      Сеанс telnet bravo приостановлен
kudos $
…
kudos $fg
telnet bravo
bravo$ logout
Connection closed by foreign host.              Подключение прервано другим хостом
kudos $
```

test

Вычисление выражения.

test ***выражение***
[выражение]

Утилита test вычисляет выражение и возвращает код условия, показывающий, что выражение либо истинно — *true* (0), либо ложно — *false* (не 0). Вместо использования слова test вокруг выражения можно поставить квадратные скобки — [](второй формат).

Аргументы

Выражение состоит из одного или нескольких критериев (см. следующий далее список), которые вычисляются утилитой test. Разделяющий два критерия оператор -a является булевым оператором И (AND): чтобы утилита test вернула код условия *true*, оба критерия должны иметь значение *true*. Оператор -o является булевым оператором ИЛИ (OR). Когда оператор -o разделяет два критерия, то для того чтобы утилита test вернула код условия *true*, значение *true* должно быть у одного или у другого критерия (или у них обоих).

Любой критерий можно превратить в его противоположность, поставив перед ним восклицательный знак (!). С помощью круглых скобок критерии можно группировать. Если круглые скобки не используются, то оператор -a имеет приоритет над оператором -o. Операторы с равным приоритетом утилита test вычисляет слева направо.

Специальные символы внутри выражений, например круглые скобки, нужно цитировать, чтобы оболочка не интерпретировала их, а передавала утилите test без изменений.

Поскольку внутри выражения каждый элемент из разряда критерия, строки или переменной является отдельным аргументом, каждый элемент нужно отделять от всех остальных пробелом. В табл. V.29 перечислены критерии, которые могут использоваться внутри выражения. А в табл. V.30 перечислены операторы отношений, используемые утилитой test.

Таблица V.29. Критерии

Критерий	Значение
строка	*True*, если *строка* имеет длину больше нуля. (Linux) *True*, если *строка* ненулевая. (OS X)
`-n` *строка*	*True*, если *строка* имеет длину больше нуля
`-z` *строка*	*True*, если *строка* имеет длину равную нулю
строка1 `=` *строка2*	*True*, если *строка1* равна *строке2*
строка1 `!=` *строка2*	*True*, если *строка1* не равна *строке2*
целое_число1 relop целое_число2	*True*, если *целое_число1* имеет указанное алгебраическое отношение к *целому_числу2*. Аргумент *relop* является оператором отношения из табл. V.30. В отдельных случаях вместо *целого_числа1 или целого_числа2* может использоваться выражение `-l строка`, выдающее длину строки
файл1 `-ef` *файл2*	*True*, если *файл1* и *файл2* имеют отношение к одному и тому же устройству и имеют одинаковые номера i-узлов
файл1 `-nt` *файл2*	*True*, если *файл1* был изменен после *файл2* (у *файла1* время последнего изменения новее, чем у *файла2*)
файл1 `-ot` *файл2*	*True*, если *файл1* был изменен перед *файлом2* (у *файла1* время последнего изменения позднее, чем у *файла2*)
`-b` *имя_файла*	*True*, если файл по имени *имя_файла* существует и является специальным блочным файлом and is a block special file
`-c` *имя_файла*	*True*, если файл по имени *имя_файла* существует и является специальным символьным файлом
`-d` *имя_файла*	*True*, если файл по имени *имя_файла* существует и является каталогом
`-e` *имя_файла*	*True*, если файл по имени *имя_файла* существует
`-f` *имя_файла*	*True*, если файл по имени *имя_файла* существует и является обычным файлом
`-g` *имя_файла*	*True*, если файл по имени *имя_файла* существует, и его бит setgid (см. раздел «Права setuid и setgid» на с. 118) установлен
`-G` *имя_файла*	*True*, если файл по имени *имя_файла* существует и связан с группой, которая является исходной группой того пользователя, который запустил команду (с тем же действующим ID группы)
`-k` *имя_файла*	*True*, если файл по имени *имя_файла* существует, и его ограничения прав на удаление (sticky bit) установлены
`-L` *имя_файла*	*True*, если файл по имени *имя_файла* существует, и является символьной ссылкой
`-O` *имя_файла*	*True*, если файл по имени *имя_файла* существует и принадлежит пользователю, запустившему команду (с тем же действующим ID пользователя)
`-p` *имя_файла*	*True*, если файл по имени *имя_файла* существует и является именованным каналом
`-r` *имя_файла*	*True*, если файл по имени *имя_файла* существует, и у пользователя, запустившего команду, есть права на чтение этого файла
`-s` *имя_файла*	*True*, если файл по имени *имя_файла* существует и содержит информацию (имеет размер больше нуля байт)
`-t` *описатель_файла*	*True*, если *описатель_файла* открыт и связан с экраном или с клавиатурой. *Описатель_файла* для стандартного ввода имеет значение 0, для стандартного вывода — 1, а для стандартной ошибки — 2

продолжение ⇙

Таблица V.29 (*продолжение*)

Критерий	Значение
-u *имя_файла*	*True*, если файл по имени *имя_файла* существует и его бит setuid (см. раздел «Права setuid и setgid» на с. 118) установлен
-w *имя_файла*	*True*, если файл по имени *имя_файла* существует и у вас имеются права на запись в него
-x *имя_файла*	*True*, если файл по имени *имя_файла* существует и у вас имеются права на его выполнение (просмотр)

Таблица V.30. Операторы отношений

Оператор отношения	Значение
-eq	Равно
-ge	Больше чем или равно
-gt	Больше чем
-le	Меньше чем или равно
-lt	Меньше чем
-ne	Не равно

Примечания

Команда test является встроенной командой оболочек Bourne Again и TC Shells.

Примеры

В следующих примерах демонстрируется работа утилиты test в сценариях оболочки Bourne Again Shell. Хотя утилита test работает и из командной строки, все же чаще всего она используется в сценариях оболочки для проверки введенных данных или прав доступа к файлу.

В первом примере выводится приглашение, строка ввода считывается в переменную, и используется синоним команды test, [], чтобы узнать, не ввел ли пользователь слово да:

```
$ cat user_in
echo -n "Введите да или нет: "
read user_input
if [ "$user_input" = "да" ]
    then
        echo Вы ввели "да".
fi
```

В следующем примере выводится приглашение на ввод имени файла, а затем используется синоним команды test, [], чтобы узнать, имеются ли у пользователя права на чтение файла (-r) и (-a) содержит ли файл информацию (-s):

```
$ cat validate
echo -n "Введите имя файла: "
read filename
if [ -r "$filename" -a -s "$filename" ]
```

```
    then
        echo Файл $filename существует и содержит информацию.
        echo У вас имеются права на чтение этого файла.
fi
```

Критерий `-t 1` проверяет, отправляет ли процесс, в котором запущена команда test, свой стандартный вывод на экран. Если отправляет, утилита test возвращает значение *true* (0). Оболочка сохраняет статус выхода из последней, запущенной ею команды в переменной `$?`. Следующий сценарий проверяет, выводится ли его информация на терминал:

```
$ cat term
test -t 1
echo "Эта программа отправляет (=0) или не отправляет (=1)
свой вывод на терминал:" $?
```

Сначала сценарий term запущен с выводом информации на терминал:

```
$ term
Эта программа отправляет (=0) или не отправляет (=1)
свой вывод на терминал: 0
```

В следующем примере запускается сценарий term, чей вывод перенаправляется в файл. Содержимое файла temp показывает, что сценарий test вернул значение 1, свидетельствующее о том, что информация выводится не на терминал.

```
$ term > temp
$ cat temp
Эта программа отправляет (=0) или не отправляет (=1)
свой вывод на терминал: 1
```

top

Динамический вывод состояния процесса.

top [ключи]

Утилита top выводит информацию о состоянии локальной системы, включая сведения о текущих процессах.

Ключи

Хотя при использовании утилиты top ставить дефисы перед ключами не требуется, но их лучше все же ставить для большей ясности и совместимости с другими утилитами. Утилиту top можно заставить работать, как будто ей был указан любой из ключей, выдавая ей команды в процессе работы. Дополнительные сведения даны в подразделе «Комментарии».

`-ca` Принуждение top работать в режиме накопления. В этом режиме производится накопительный подсчет времени и событий. Если нужно использовать ключ `-c` для запуска утилиты top в другом режиме, обратитесь к man-странице, посвященной этой утилите. (OS X)

`-d` *ss.tt* (delay — задержка). Указание *ss.tt* в качестве количества секунд и десятых долей секунд задержки от одного вывода обновления до другого. Значение по умолчанию составляет 3 секунды. (Linux)

`-i` Игнорирование неработающих и зомби-процессов (процессов, у которых отсутствует родительский процесс). (Linux)

`-n` *n* (number — количество). Указание количества итераций: утилита top обновляет вывод *n* раз и завершает свою работу. (Linux)

`-p` *n* (PID). Мониторинг процесса с PID-номером *n*. Этот ключ можно использовать в командной строке до 20 раз или указать *n* в виде списка, содержащего до 20 PID-номеров, разделенных запятыми. (Linux)

`-S` (sum — сумма). Принуждение top работать в накопительном режиме. В этом режиме время использования центрального процессора, выводимое для процесса, включает время ответвленных от него процессов, и время использования центрального процессора, накопленное уже завершившими свою работу дочерними процессами. (Linux)

`-s` (secure — безопасность). Запуск top в безопасном режиме, ограничение количества доступных для использования в процессе работы top команд теми из них, которые представляют наименьший риск с точки зрения безопасности. (Linux)

`-s` *ss* (seconds — секунды). Указание *ss* в качестве количества секунд задержки от одного вывода обновления до другого. Значение по умолчанию составляет одну секунду. (OS X)

Комментарии

В первых нескольких строках утилита top выводит суммарные сведения о состоянии локальной системы. Вывод любой из этих строк можно включить или выключить с помощью переключателей (интерактивных командных клавиш), указанных в следующем ниже описании. В первой строке выводится информация, аналогичная той, которую выводит утилита uptime, и показывается текущее время, количество времени, прошедшего с момента последней загрузки локальной системы, количество вошедших в систему пользователей и средние показатели загруженности за последнюю минуту, 5 и 15 минут (включается и выключается клавишей l (буква «l» в нижнем регистре)). Во второй строке показывается количество запущенных процессов (включается и выключается клавишей t). Последние три строки выводят сведения об использовании центрального процессора (также включается и выключается клавишей t), памяти (включается и выключается клавишей m) и свопируемого пространства (также включается и выключается клавишей m).

Вся остальная часть выводимой информации относится к сведениям об отдельных процессах, перечисленных в порядке убывания времени использования центрального процессора (то есть сначала выводятся сведения о процессе, использующем центральный процессор наиболее интенсивно). По умолчанию утилита top выводит сведения о таком количестве процессов, которое помещается на экране.

В табл. V.31 описывается значение полей, выводимых для каждого процесса.

Таблица V.31. Имена полей

Имя	Значение
PID	Идентификационный номер процесса
USER	Имя пользователя, являющегося владельцем процесса
PR	Приоритет процесса
NI	Значение, установленное утилитой nice (см. соответствующее описание)
VIRT	Количество килобайт виртуальной памяти, используемой процессом
RES	Количество килобайт физической (несвопируемой) памяти, используемой процессом
SHR	Количество килобайт совместно используемой памяти, используемой процессом
S	Состояние процесса (см. STAT в табл. V.28)
%CPU	Процент от общего количества времени использования центрального процессора, приходящегося на процесс
%MEM	Процент используемого процессом общего объема физической памяти
TIME[+]	Общее количество времени центрального процессора, использованного процессом
COMMAND	Командная строка, запустившая процесс или имя программы (переключаются с помощью команды c)

Во время работы утилиты top ей можно выдавать следующие команды, меняющие характер ее поведения:

h (help — справка). Вывод перечня команд, которыми можно пользоваться в процессе работы утилиты top. (Linux)

? (help — справка). Вывод перечня команд, которыми можно пользоваться в процессе работы утилиты top. (OS X)

k (kill — завершить). Предоставление возможности завершить работу процесса. При работе без root-прав можно завершить только тот процесс, владельцем которого вы являетесь. При использовании этой команды утилита top выводит приглашение на ввод PID-номера процесса и сигнала для отправки этому процессу. Можно ввести либо номер сигнала, либо его имя. (Список сигналов приведен в табл. 10.5.) При работе в безопасном режиме эта команда отключена. (Linux)

n (number — количество). При выдаче этой команды top просит ввести количество процессов, сведения о которых нужно вывести. Если ввести 0 (значение по умолчанию), top показывает сведения о таком количестве процессов, которое помещается на экране.

q (quit — завершить). Завершение работы утилиты top.

r (renice). Изменение приоритета запущенного процесса (см. описание утилиты renice). При работе без root-прав можно изменить приоритет только тех процессов, владельцем которых вы являетесь, и только в сторону его уменьшения путем ввода положительных значений. Пользователь, работающий с root-правами, может ввести отрицательное значение, повышая приоритет процесса. При работе в безопасном режиме эта команда отключается. (Linux)

S (switch — переключение). Переключает работу top между накопительным и обычным режимами. Подробности даны в описании ключа -S. (Linux)

s (seconds — секунды). Вывод приглашения на ввод количества секунд задержки между выводом очередного обновления информации (по умолчанию используется значение 3). Можно ввести дробную часть или 0 (для постоянного обновления). При работе под управлением Linux эта команда в безопасном режиме отключается.

W (write — запись). Запись текущей конфигурации утилиты top в свой персональный конфигурационный файл (~/.toprc). (Linux)

SPACE Обновление экрана.

Примечания

Версии утилиты top для Linux и для OS X сильно отличаются друг от друга. Несмотря на применимость к обеим версиям, данное описание утилиты top в большей степени ориентировано на версию утилиты для Linux. Дополнительные сведения о версии для OS X даны на man-странице, посвященной утилите top для OS X.

Утилита top похожа на утилиту ps, но она периодически обновляет выводимую информацию, позволяя отслеживать поведение локальной системы с течением времени.

Эта утилита показывает для каждого процесса только ту часть командной строки, которая умещается на экране. Если процесс свопирован на диск, top заменяет командную строку именем команды в круглых скобках.

При работе под управлением Linux утилита top использует файловую систему proc. Когда proc не смонтирована, утилита top не работает.

Запрос непрерывного обновления практически всегда является ошибкой. Вывод обновляется слишком быстро, а загруженность системы возрастает слишком сильно.

Пример

Следующий вывод является типичным результатом выполнения команды top:

```
top - 17:48:53 up  6:21,  1 user,  load average: 0.71, 1.00, 0.80
Tasks: 178 total,   1 running, 176 sleeping,   0 stopped,   1 zombie
Cpu(s): 16.2%us, 3.0%sy, 0.0%ni, 80.4%id, 0.0%wa, 0.3%hi, 0.0%si, 0.0%st
Mem:   8188664k total,  8122636k used,    66028k free,   127200k buffers
Swap: 11912112k total,        0k used, 11912112k free,  7217728k cached
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 7841 mark      20   0  108m  29m  12m S   15  0.4   5:14.62 npviewer.bin
 7810 mark      20   0  675m 175m  27m S    8  2.2  24:07.87 firefox
12084 mark      20   0  568m  70m  32m S    7  0.9   5:14.39 amarokapp
13428 mark      10 -10 1715m  74m  62m S    5  0.9   6:25.19 vmware-vmx
 7147 root      20   0  449m 102m  14m S    4  1.3  57:57.59 Xorg
 7676 mark      20   0  159m  17m  11m S    1  0.2   0:05.14 konsole
13437 root      20   0  300m  18m  10m S    1  0.2   0:21.14 firestarter
16542 mark      20   0 18992 1332  940 R    1  0.0   0:00.16 top
 7672 mark      20   0  323m  21m  10m S    0  0.3   1:09.47 artsd
 7747 mark      20   0  306m  45m  25m S    0  0.6   1:20.28 vmware
```

```
    1 root      20   0  4020  876  592 S    0  0.0   0:01.24 init
    2 root      15  -5     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      RT  -5     0    0    0 S    0  0.0   0:00.00 migration/0
    4 root      15  -5     0    0    0 S    0  0.0   0:03.04 ksoftirqd/0
...
```

touch

Создание файла или изменение времени последнего обращения к нему и (или) времени его последнего изменения.

touch [ключи] список_файлов

Утилита touch изменяет время последнего обращения к файлу и (или) время его последнего изменения на текущее или на указанное вами время. Утилиту touch можно также использовать для создания файла.

Аргументы

Список_файлов — это список путевых имен файлов, которые создаются или претерпевают изменения с помощью touch.

Ключи

При работе под управлением Linux утилита touch допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме специально оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X. Без ключей утилита touch изменяет время последнего обращения и время последнего изменения на текущее. Если не указывается ключ -c (--no-create), утилита touch создает те файлы, которые еще не существуют.

`-a` Обновление только времени последнего обращения, при этом время последнего изменения остается прежним.

`--no-create` `-c` Отказ от создания несуществующих файлов.

`--date=`*строка_даты* `-d` *строка_даты* Обновление показателей времени с применением даты, указанной в *строке_даты*. В *строке_даты* можно использовать большинство общеизвестных форматов. Компоненты даты и времени, не включенные в *строку_даты*, берутся из текущих даты и времени. Этот ключ не может быть использован вместе с ключом -t. (Linux)

`-m` Обновление только времени последнего изменения, при этом время последнего обращения остается прежним.

`--reference=`*файл* `-r` *файл* Обновление показателей времени с использованием тех же показателей, которые имеются у *файла*.

`-t [[cc]yy] nnddhhmm [.ss]` Изменение показателей времени на дату, указанную в аргументе. Аргумент *nn* представляет месяц (01–12), *dd* представляет день месяца (01–31), *hh* представляет час по 24-часовой шкале (00–23), а *mm* — минуты (00–59). Должны быть указаны как минимум три поля. Используя аргумент *.ss*, можно также указать количество секунд, прошедших с начала минуты.

В необязательном аргументе *cc* указываются первые две цифры года (значение века за вычетом единицы), а в аргументе *yy* указывают последние две цифры года. Если год не указан, touch использует значение текущего года. Если не задан аргумент *cc*, touch берет значение 20 для *yy* в диапазоне 0–68 и значение 19 для *yy* в диапазоне 69–99. Этот ключ не может использоваться вместе с ключом `-d`.

Примеры

В первых трех командах показано, как с помощью утилиты touch обновляется существующий файл. Утилита ls с ключом `-l` выводит время последнего изменения файла. В последних трех командах показывается, как touch создает файл.

```
$ ls -l program.c
-rw-r--r--   1 max group 5860 Apr 21 09:54 program.c
$ touch program.c
$ ls -l program.c
-rw-r--r--   1 max group 5860 Aug 13 19:01 program.c
$ ls -l read.c
ls: read.c: No such file or directory Файл или каталог с таким именем не существует
$ touch read.c
$ ls -l read.c
-rw-rw-r--   1 max group 0 Aug 13 19:01 read.c
```

Первая из следующих команд ls показывает время последнего изменения файлов, а вторая команда ls (с ключами `-lu` options) показывает время последнего обращения к файлам:

```
$ ls -l
-rw-r--r-- 1 max group  552 Jan 10 19:44 cases
-rw-r--r-- 1 max group 2209 Apr 18 04:24 excerpts
$ ls -lu
-rw-r--r-- 1 max group  552 May 29 15:28 cases
-rw-r--r-- 1 max group 2209 May 29 15:28 excerpts
```

Следующий пример работает с двумя файлами, показанными выше, и демонстрирует использование ключа `-a` для изменения только лишь времени последнего обращения к файлам и использование ключа –t для указания даты, которую touch использует вместо текущих показаний даты и времени. После выполнения команды touch команда ls показывает, что время последнего обращения к файлам cases и excerpts обновилось, но время их последних изменений осталось прежним.

```
$ touch -at 02040608 cases excerpts

$ ls -l
-rw-r--r-- 1 max group  552 Jan 10 19:44 cases
-rw-r--r-- 1 max group 2209 Apr 18 04:24 excerpts
```

```
$ ls -lu
-rw-r--r-- 1 max group  552 Feb  4 06:08 cases
-rw-r--r-- 1 max group 2209 Feb  4 06:08 excerpts
```

tr

Замена указанных символов.

tr [ключи] строка1 [строка2]

Утилита tr читает данные со стандартного ввода и для каждого введенного символа либо преобразует его в альтернативный символ, удаляет символ или оставляет его в неизменном виде. Эта утилита читает данные со стандартного ввода и записывает их на стандартный вывод.

Аргументы

Утилита tr обычно использует два аргумента, ***строка1*** и ***строка2***. Позиция каждого символа в этих двух строках играет важную роль: при каждом обнаружении утилитой tr в своем вводе символа из ***строки1***, она заменяет этот символ соответствующим символом из ***строки2***.

Если указан один аргумент, ***строка1***, и ключ `-d` (`--delete`), tr удаляет символы, указанные в ***строке1***. Ключ `-s` (`--squeeze-repeats`) приводит к замене нескольких последовательно идущих символов из тех, что указаны в ***строке1***, одним символом (например, `abbc` превращается в `abc`).

Диапазоны

Диапазон символов похож по своему действию на класс символов внутри регулярного выражения. GNU-версия утилиты tr не поддерживает диапазоны (классы символов), заключенные в квадратные скобки. Диапазон символов можно указать, поставив после символа, следующего раньше в сортирующей последовательности, дефис и символ, который в сортирующей последовательности следует позже. Например, запись `1-6` расширяется в `123456`. Хотя в ASCII диапазон `A-Z` расширяется вполне предсказуемым образом, этот подход не будет работать при использовании сортирующей последовательности EBCDIC, поскольку эти символы не являются последовательными в EBCDIC. Решение этой проблемы рассмотрено в подразделе «Классы символов».

Классы символов

Класс символов, используемый для утилиты tr, не похож на тот класс символов, который рассматривается в остальных частях данной книги. (Для понятия *класс символов*, используемого в данной книге, GNU-документация использует термин *оператор списка*.) Определение класса символов дается в формате *'[:класс:]'*, где ***класс*** — это один из классов символов из табл. V.32. Класс символов нужно указывать в ***строке1*** (но не в ***строке2***), если только не выполняется конвертация регистров (см. подраздел «Примеры») или не используются вместе ключи `-d` и `-s`.

Таблица V.32. Классы символов

Класс	Значение
alnum	Буквы и цифры
alpha	Буквы
blank	Пробельные символы
cntrl	Управляющие (CONTROL) символы
digit	Цифры
graph	Символы, выводимые на печать, за исключением пробелов
lower	Символы в нижнем регистре
print	Символы, выводимые на печать, включая пробелы
punct	Символы пунктуации
space	Горизонтальный или вертикальный пробельный символ
upper	Символы в верхнем регистре
xdigit	Шестнадцатеричные цифры

Ключи

Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме специально оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--complement` `-c` Дополнение ***строки1***, принуждение tr искать соответствие всем символам, за исключением тех, что указаны в ***строке1***.

`--delete -d` Удаление символов, соответствующих символам, указанным в ***строке1***. При использовании этого ключа с ключом `-s` (`--squeeze-repeats`) нужно указать обе строки: ***строку1*** и ***строку2*** (см. подраздел «Примечания»).

`--help` Вывод краткой справки по использованию утилиты tr, включая специальные символы, которые могут использоваться в ***строке1*** и ***строке2***. (Linux)

`--squeeze-repeats -s` Замена нескольких последовательно идущих символов из тех, что указаны в ***строке1***, одним символом, когда tr вызывается только с одним строковым аргументом. Если используются оба аргумента, ***строка1*** и ***строка2***, утилита tr сначала переводит символы, встречающиеся в ***строке1***, в символы, соответствующие этим символам в ***строке2***, а затем заменяет несколько последовательно идущих символов из тех, что указаны в ***строке2***, одним символом.

`--truncate-set1 -t` Усечение ***строки1*** до длины ***строки2*** перед обработкой ввода. (Linux)

Примечания

Когда ***строка1*** длиннее ***строки2***, для преобразования используется начальная часть ***строки1*** (равная по длине ***строке2***). Если ***строка1*** короче ***строки2***, tr использует последний символ ***строки1***, чтобы расширить ***строку1*** до размера ***строки2***. В этом случае tr отступает от стандарта POSIX, который не определяет результат.

При одновременном использовании ключа -d (--delete) и ключа -s (--squeeze-repeats) tr сначала удаляет те символы, которые указаны в ***строке1***, а затем заменяет несколько последовательно идущих символов из тех, что указаны в ***строке2***, одним символом.

Примеры

Для представления диапазона символов в ***строке1*** и ***строке2*** можно воспользоваться дефисом. Две командные строки в следующем примере приводят к одинаковому результату:

```
$ echo abcdef | tr  'abcdef' 'xyzabc'
xyzabc
$ echo abcdef | tr  'a-f' 'x-za-c'
xyzabc
```

В следующем примере демонстрируется популярный метод маскировки текста, часто называемый ROT13 (rotate 13 — вращение 13), поскольку он заменяет первую букву алфавита тринадцатой, вторую — четырнадцатой и т. д.

```
$ echo The punchline of the joke is … |
> tr 'A-M N-Z a-m n-z' 'N-Z A-M n-z a-m'
Gur chapuyvar bs gur wbxr vf …
```

Чтобы вернуть этому тексту смысл, нужно изменить порядок следования аргументов для утилиты tr:

```
$ echo Gur chapuyvar bs gur wbxr vf … |
> tr 'N-Z A-M n-z a-m' 'A-M N-Z a-m n-z'
The punchline of the joke is …
```

Ключ --delete заставляет tr удалять выбранные символы:

```
$ echo If you can read this, you can spot the missing vowels! |
> tr --delete 'aeiou'
If y cn rd ths, y cn spt th mssng vwls!
```

В следующем примере tr заменяет символы и сокращает пары одинаковых символов до одиночных символов:

```
$ echo tennessee | tr -s 'tnse' 'srne'
serene
```

В следующем примере каждая последовательность неалфавитных символов (дополнение до всех алфавитных символов, указанное с помощью класса символов alpha) в файле draft1 заменяется одним символом разделителя строк (NEWLINE). При этом выводится список слов, по одному слову на каждой строке.

```
$ tr -c -s '[:alpha:]' '\n' < draft1
```

В следующем примере используется класс символов для преобразования в верхний регистр строки hi there:

```
$ echo hi there | tr '[:lower:]' '[:upper:]'
HI THERE
```

tty

Вывод имени терминала.

tty [ключ]

Утилита tty выводит имя стандартного ввода, если в его качестве используется терминал, а в противном случае выводит сообщение `not a tty` (не tty). Статус выхода равен нулю, если используется терминал, и единице, если используется не терминал.

Аргументы

Аргументы с этой утилитой не используются.

Ключи

При работе под управлением Linux утилита tty допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--silent` *или* `--quiet` *или* `-s` — Принуждение утилиты tty к отказу от вывода информации. При этом устанавливает статус выхода tty.

Примечания

Термин tty является сокращением от teletypewriter, терминального устройства, с помощью которого впервые была запущена система UNIX. Эта команда появилась в UNIX, а в Linux она сохранилась для поддержки совместимости и в силу традиции.

Примеры

Использование tty показано в следующем примере:

```
$ tty
/dev/pts/11
$ echo $?
0
$ tty < memo
not a tty
$ echo $?
1
```

tune2fs

Изменение параметров файловой системы ext2, ext3 или ext4.

tune2fs [ключи] устройство

Утилита tune2fs выводит и изменяет параметры файловой системы на файловых системах ext2, ext3 и ext4. Эта утилита может также установит журналирование на

файловой системе ext2, превратив ее в файловую систему ext3. При обычных правах доступа к файловой системе утилита tune2fs должна быть запущена пользователем, работающим с root-правами. Утилита tune2fs доступна только при работе под управлением Linux. (Linux)

Аргументы

Устройство — это имя устройства, например /dev/sda8, содержащего файловую систему, параметры которой нужно вывести или изменить.

Ключи

`-C` *n* (count — количество). Задание с помощью *n* количества установок файловой системы без проведения проверки. Этот ключ применяется для установки периодичности проверок файловой системы (см. подраздел «Комментарии») и для того, чтобы заставить провести проверку при ее следующей загрузке.

`-c` *n* (max count — максимальное количество). Задание с помощью *n* максимального количества установок файловой системы между ее проверками. Чтобы проигнорировать этот параметр, нужно установить для *n* значения `0` (нуль).

`-e` *поведение* (error — ошибка). Указание ядру порядка действий при обнаружении ошибки. *Поведение* нужно установить в `continue` (продолжение работы), `remount-ro` (переустановка файловой системы в режим только для чтения) или `panic` (вызов для ядра состояния тревоги). Независимо от значения, использованного при установке этого ключа, ошибка заставит утилиту fsck проверить файловую систему при следующей загрузке операционной системы.

`-i` *n[u]* (interval — интервал). Установка максимального времени между проверками файловой системы на *n* периодов времени. Если аргумент *u* не используется или если он имеет значение `d`, период времени устанавливается в днях. Для использования в этом качестве недель следует установить для *u* значение `w`, а для использования месяцев следует установить значение `m`. Чтобы проигнорировать этот параметр, нужно установить для *n* значение `0` (нуль). Поскольку проверка файловой системы проводится только в период загрузки операционной системы, время, определенное этим параметром, может быть превышено.

`-j` (journal — журналирование). Добавление журналирования, которое проводится в ext3, в файловую систему ext2.

`-l` (list — перечень). Вывод перечня информации о файловой системе.

`-T` *дата* (time — время). Установка времени последней проверки файловой системы в соответствии в аргументом *дата*. Этот аргумент представляет собой время и дату в формате *yyyynndd[hh[mm]ss]]]*. Здесь поле *yyyy* представляет год, *nn* — номер месяца (01–12), а *dd* — день месяца (01–31). Эти поля нужно указать в обязательном порядке. Поле *hh* представляет час по 24-часовой шкале (00–23), *mm* — минуты (00–59), а *.ss* — количество секунд, прошедших с начала минуты. Можно также указать *дату* как `now` (сейчас).

Комментарии

Проверка большой файловой системы может занять много времени. Если одновременно проводится проверка всех файловых систем, загрузка операционной системы может проходить очень медленно. Чтобы разнести по времени проверки файловых систем и исключить их проведение в одно и то же время, следует воспользоваться ключами `-C` и (или) `-T`.

Пример

При запуске утилиты tune2fs с ключом `-l` на обычной файловой системе ext3 выводится примерно следующая информация:

```
# /sbin/tune2fs -l /dev/sda1
tune2fs 1.41.4 (27-Jan-2009)
Filesystem volume name:   <none>
Last mounted on:          <not available>
Filesystem UUID:          b6d9714e-ed5d-45b8-8023-716a669c16d8
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_index
filetype needs_recovery sparse_super large_file
Filesystem flags:         signed_directory_hash
Default mount options:    (none)
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              624624
Block count:              2498099
Reserved block count:     124904
Free blocks:              1868063
Free inodes:              509355
First block:              0
Block size:               4096
Fragment size:            4096
Reserved GDT blocks:      609
Blocks per group:         32768
Fragments per group:      32768
Inodes per group:         8112
Inode blocks per group:   507
Filesystem created:       Mon Apr 27 09:41:43 2009
Last mount time:          Tue May  5 03:54:59 2009
Last write time:          Tue May  5 03:54:59 2009
Mount count:              4
Maximum mount count:      31
Last checked:             Mon Apr 27 09:41:43 2009
Check interval:           15552000 (6 months)
Next check after:         Sat Oct 24 09:41:43 2009
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
First inode:              11
```

```
Inode size:              256
Required extra isize:    28
Desired extra isize:     28
Journal inode:           8
First orphan inode:      308701
Default directory hash:  half_md4
Directory Hash Seed:     bceae349-a46f-4d45-a8f1-a21b1ae8d2bd
Journal backup:          inode blocks
```

umask

Установка маски прав доступа, используемой при создании файла.

*umask [**маска**]*

Встроенная команда umask определяет ***маску***, используемую системой для установки прав доступа к файлу при его создании. Эта встроенная команда работает в разных оболочках немного по-разному.

Аргументы

Маска должна быть трехразрядным восьмеричным числом (в оболочках bash и tcsh) или символьным значением (bash), аналогичным используемому с утилитой chmod (см. описание этой утилиты). ***Маска*** определяет те права доступа, которые не предоставляются.

Когда ***маска*** представлена в виде восьмеричного числа, цифры соответствуют правам для владельца файла, для участника группы, с которой связан файл, и для всех остальных. Поскольку ***маска*** определяет непредоставляемые права, система, когда вы создаете файл, вычитает каждую из этих цифр из 7. Получившиеся три восьмеричных числа определяют права доступа к файлу (это те самые числа, которые используются при работе с утилитой chmod). ***Маска***, указанная в виде символьного значения, также определяет непредоставляемые права. См. подраздел «Примечания».

Без аргументов встроенная команда umask выводит маску, используемую при установке прав доступа к создаваемому файлу.

Ключ

`-S` (symbolic — символьная). Вывод маски прав доступа к создаваемому файлу в символьном виде.

Примечания

Большинство утилит и приложений не предпринимают попыток создания файлов с правами на их выполнение, независимо от того значения, которое указано в ***маске***; они заранее предполагают, что исполняемый файл создавать не нужно. В результате этого, когда утилита или приложение, например утилита touch, создает файл, система вычитает каждую цифру в ***маске*** из числа 6. Исключение составляет утилита mkdir, которая предполагает, что вам нужен установленный бит выполнения (доступ в случае работы с каталогом). См. подраздел «Примеры».

Программа umask является встроенной командой оболочек bash и tcsh и, как правило, обращается к файлу инициализации для вашей оболочки (~/.bash_profile для bash или ~/.login для tcsh).

При работе в bash аргумент u=rwx,go=r сбрасывает все биты в ***маске*** для владельца, и сбрасывает бит права на чтение в ***маске*** для участников группы и всех остальных пользователей (***маска*** получает значение 0033), заставляя эти биты быть установленными при определении прав доступа к файлу (744 или 644). Дополнительные сведения о символьном режиме указания прав доступа даны в описании утилиты chmod.

Примеры

Следующие команды устанавливают маску прав доступа к создаваемым файлам и показывают значение этой маски и ее действие при создании файла и каталога. ***Маска***, имеющая значение 022, после вычитания из 777 дает права доступа 644 (rw-r--r--) для файла и 755 (rwxr-xr-x) для каталога:

```
$ umask 022
$ umask
0022
$ touch afile
$ mkdir adirectory
$ ls -ld afile adirectory
drwxr-xr-x  2 max max 4096 Jul 24 11:25 adirectory
-rw-r--r--  1 max max 0 Jul 24 11:25 afile
```

В следующем примере та же ***маска*** устанавливается с помощью символьных значений. Ключ -S приводит к выводу ***маски*** в символьном режиме:

```
$ umask u=rwx,g=rx,o=rx
$ umask
0022
$ umask -S
u=rwx,g=rx,o=rx
```

uniq

Вывод уникальных строк.

uniq [ключи] [файл_ввода] [файл_вывода]

Утилита uniq выводит свой ввод, удаляя все лишние копии последовательно повторяющихся строк. Если файл был отсортирован (см. описание утилиты sort), утилита uniq обеспечивает отсутствие в выводе двух одинаковых строк.

Аргументы

Когда ***файл_ввода*** не указан, утилита uniq осуществляет чтение со стандартного ввода. Если не указан ***файл_вывода***, uniq ведет запись в стандартный вывод.

Ключи

При работе под управлением Linux утилита uniq допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме специально оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X. Под *полем* понимается последовательность символов, окруженная пробелами, символами табуляции, символами разделителей строк или комбинацией из этих символов.

`--count -c` Установка перед каждой строкой количества ее появлений во входном файле.

`--repeated -d` Вывод одной копии повторяющихся строк без вывода тех строк, которые не повторяются.

`--skip-fields=`*nfield* `-f` *nfield* Игнорирование первых *nfield* полей, отделенных друг от друга пробельными символами. Утилита uniq берет за основу своих сравнений оставшуюся часть строки, включая ведущие пробельные символы следующего поля строки (см. ключ `-s` (`--skip-chars`)).

`--ignore-case -i` Игнорирование регистра букв в сравниваемых строках.

`--skip-chars=`*nchar* `-s` *nchar* Игнорирование первых *nchar* символов в каждой строке. Если вдобавок воспользоваться ключом `-f` (`--skip-fields`), то утилита uniq проигнорирует первые *nfield* поля, за которыми следуют *nchar* символов. Этот ключ можно использовать для перехода через начальные пробельные символы поля.

`--unique -u` Вывод только тех строк, которые *не* повторяются.

`--check-chars=`*nchar* `-w` *nchar* Проведение сравнения вплоть до *nchars* символов в строке после ее обработки, связанной с применением ключей `-f` (`--skip-fields`) и `-s` (`--skip-chars`). По умолчанию утилита uniq проводит сравнение всей строки. (Linux)

Примеры

В этих примерах предполагается, что файл по имени test находится в рабочем каталоге и содержит следующий текст:

```
$ cat test
boy took bat home
boy took bat home
girl took bat home
dog brought hat home
dog brought hat home
dog brought hat home
```

Без использования ключей утилита uniq выводит только одну копию последовательно повторяющихся строк:

```
$ uniq test
boy took bat home
```

```
girl took bat home
dog brought hat home
```

Ключ `-c` (`--count`) приводит к выводу количества последовательных появлений каждой строки в файле:

```
$ uniq -c test
   2 boy took bat home
   1 girl took bat home
   3 dog brought hat home
```

Ключ `-d` (`--repeated`) приводит к выводу только тех строк, которые последовательно повторяются в файле:

```
$ uniq -d test
boy took bat home
dog brought hat home
```

Ключ `-u` (`--unique`) приводит к выводу только тех строк файла, которые не имеют последовательных повторений:

```
$ uniq -u test
girl took bat home
```

В следующем примере ключ `-f` (`--skip-fields`) приводит к пропуску первого поля в каждой строке, заставляя строки, начинающиеся со слова `boy`, и строку, начинающуюся со слова `girl`, считаться последовательно повторяющимися строками. Утилита **uniq** выводит только один экземпляр этих строк:

```
$ uniq -f 1 test
boy took bat home
dog brought hat home
```

В следующем примере используются оба ключа `-f` (`--skip-fields`) и `-s` (`--skip-chars`), сначала для пропуска двух полей, а затем для пропуска двух символов. Два символа, пропускаемые этой командой, включают пробел, разделяющий второе и третье поля, и первый символ третьего поля. При игнорировании этих символов все строки, содержащие подстроку `at home`, считаются последовательно повторяющимися. Утилита **uniq** выводит только первую из этих строк:

```
$ uniq -f 2 -s 2 test
boy took bat home
```

w

Вывод информации о пользователях системы.

w [ключи] [имя_пользователя]

Утилита **w** выводит имена пользователей, вошедших в систему, а также номера их терминальных устройств, показания времени их входа в систему, команды, ими запущенные, и другую информацию.

Аргументы

Имя_пользователя ограничивает вывод только той информацией, которая имеет отношение к данному пользователю.

Ключи

`-f` (from — из). Удаление столбца FROM. Для пользователей, подключившихся напрямую, в этом поле ставится прочерк. (Linux)

`-h` (no header — без заголовка). Подавление вывода строки заголовка.

`-s` (short — краткий). Вывод меньшего объема информации: имени пользователя, названия терминального устройства, времени простоя и команды. (Linux)

Комментарии

В первой строке утилита w выводит такую же информацию, которую выводит утилита uptime. Эта строка включает время дня, продолжительность работы системы (в днях, часах и минутах), количество вошедших в систему пользователей и сведения о занятости системы (среднюю загруженность). Слева направо средняя загруженность показывает количество процессов, ожидающих запуска за последние 1, 5 и 15 минут.

Информационные столбцы, выводимые утилитой w, имеют следующие заголовки:

```
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
```

В столбце USER содержится имя пользователя. В столбце TTY содержится имя устройства для той линии, по которой пользователь вошел в систему. В столбце FROM содержится имя системы, с которой вошел в систему удаленный пользователь, для локальных пользователей в нем ставится прочерк. В столбце LOGIN@ содержится дата и время входа пользователя в систему. В столбце IDLE содержится количество минут, прошедших со времени последней работы пользователя на клавиатуре. В столбце JCPU содержится время загрузки центрального процессора всеми процессами, прикрепленными к пользовательскому терминалу, за исключением уже завершившихся фоновых заданий. В столбце PCPU содержится время, занятое процессом, названным в столбце WHAT. В столбце WHAT содержится команда, запущенная пользователем.

Примеры

В первом примере показывается полный список, создаваемый утилитой w:

```
$ w
 10:26am  up 1 day, 55 min,  6 users,  load average: 0.15, 0.03, 0.01
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
max      tty1     -                 Fri 9am 20:39m  0.22s  0.01s  vim td
max      tty2     -                 Fri 5pm 17:16m  0.07s  0.07s  -bash
root     pts/1    -                 Fri 4pm 14:28m  0.20s  0.07s  -bash
sam      pts/2    -                 Fri 5pm  3:23   0.08s  0.08s  /bin/bash
hls      pts/3    potato            10:07am  0.00s  0.08s  0.02s  w
```

В следующем примере ключ -s заставляет создать сокращенный список:

```
$ w -s
 10:30am  up 1 day, 58 min,  6 users,  load average: 0.15, 0.03, 0.01
USER     TTY      FROM              IDLE  WHAT
max      tty1     -                20:43m  vim td
max      tty2     -                17:19m  -bash
root     pts/1    -                14:31m  -bash
```

```
sam      pts/2    -                0.20s  vim memo.030125
hls      pts/3    potato           0.00s  w -s
```

В последнем примере запрашивается информация, касающаяся только пользователя Max:

```
$ w max
 10:35am  up 1 day,  1:04,  6 users,  load average: 0.06, 0.01, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
max      tty1     -                 Fri 9am 20:48m  0.22s  0.01s  vim td
max      tty2     -                 Fri 5pm 17:25m  0.07s  0.07s  -bash
```

wc

Вывод количества строк, слов и байтов.

wc [ключи] [список_файлов]

Утилита wc выводит количество строк, слов и байтов, имеющихся в ее вводе. Если в командной строке указывается более одного файла, wc выводит суммарное количество для каждого файла, а также общее количество для всех файлов.

Аргументы

Список_файлов — это список путевых имен одного или нескольких файлов, которые подвергаются анализу с помощью утилиты wc. Если ***список_файлов*** не указывается, утилита wc получает данные со стандартного ввода.

Ключи

При работе под управлением Linux утилита wc допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме специально оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--bytes -c` Вывод только количества байт, имеющихся во вводе.

`--max-line-length -L` Вывод длины самой длинной строки ввода. (Linux)

`--lines -l` (буква «l» в нижнем регистре). Вывод только количества строк (то есть количества символов разделителей строк), имеющихся во вводе.

`--chars -m` Вывод только количества символов, имеющихся во вводе.

`--words -w` Вывод только количества слов, имеющихся во вводе.

Примечания

Слово — это последовательность символов, ограниченная пробелами, символами табуляции, разделителями строк или комбинациями этих символов.

Примеры

Следующая команда анализирует содержимое файла по имени memo.Числа в выводе представляют количество строк, слов и байтов в файле.

```
$ wc memo
      5      31      146 memo
```

Следующая команда выводит количество строк и слов в трех файлах. В самой последней строке, содержащей в самом правом столбце слово `total` (всего), содержится итоговая сумма по каждому столбцу.

```
$ wc -lw memo1 memo2 memo3
     10      62 memo1
     12      74 memo2
     12      68 memo3
     34     204 total
```

which

Вывод места в переменной PATH, где содержится команда.

which ***список_команд***

Для каждой команды в ***списке_команд*** утилита which ищет каталоги в переменной PATH (см. раздел «Переменная PATH: Указание тех мест, где оболочка ищет программы» на с. 312) и выводит абсолютное имя первого найденного файла, простое имя которого совпадает с названием команды.

Аргументы

Список_команд — это список, состоящий из одной или нескольких команд (утилит), которые ищутся с помощью утилиты which. Для каждой команды утилита which ведет последовательный поиск каталогов, перечисленных в переменной среды окружения PATH, и выводит полное имя первой найденной команды (исполняемого файла). Если which не находит команды, она выводит соответствующее сообщение.

Ключи

Ключи, перед которыми ставится двойной дефис (`--`), работают только под управлением Linux. Кроме специально оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--all -a` Вывод сведений обо всех соответствующих исполняемых файлах, пути к которым указаны в переменной PATH, а не только о первом из них.

`--read-alias -i` Чтение псевдонимов со стандартного ввода и вывод отчета о соответствующих псевдонимах в дополнение к сведениям об исполняемых файлах, пути к которым указаны в переменной PATH (выключается с помощью ключа `--skip-alias`). (Linux)

`--read-functions` Чтение функций оболочки со стандартного ввода и вывод отчета о соответствующих функциях в дополнение к сведениям об исполняемых файлах, пути к которым указаны в переменной `PATH` (выключается с помощью ключа `--skip-functions`). (Linux)

`--show-dot` Вывод вместо абсолютного имени символов `./`, когда каталог в переменной `PATH` начинается с точки и соответствует исполняемому файлу в этом каталоге (выключается с помощью ключа `--skip-dot`). (Linux)

`--show-tilde` Вывод символа тильды (~) вместо абсолютного имени личного каталога пользователя, если это приемлемо. Этот ключ игнорируется, когда утилита which запускается пользователем, работающим с root-правами. (Linux)

`--tty-only` Отказ от обработки дополнительных ключей (справа от этого ключа), если процесс, в котором запущена утилита which, не прикреплен к терминалу. (Linux)

Примечания

В некоторых дистрибутивах для утилиты which определен следующий псевдоним:

```
$ alias which
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
```

Если утилита which ведет себя неожиданным для вас образом, проверьте, не запущен ли вами псевдоним. Предыдущий псевдоним заставляет утилиту which работать только в интерактивном режиме (`--tty-only`) и показывать псевдонимы, выводить рабочий каталог в виде точки, когда это приемлемо, а также показывать личный каталог пользователя в виде тильды.

Оболочка TC Shell включает в себя встроенную команду which (см. man-страницу, посвященную tcsh), которая работает немного иначе, чем утилита which (/usr/bin/which). Без использования ключей утилита which не определяет местонахождения псевдонимов, функций и встроенных команд оболочки, поскольку они не фигурируют в значении переменной `PATH`. В отличие от утилиты встроенная в tcsh команда which показывает, где находятся псевдонимы, функции и встроенные команды оболочки.

Примеры

В первом примере процитирована первая буква команды (`\which`), чтобы не дать оболочке вызвать псевдоним (см. главу 8, совет «Как удержать оболочку от вызова псевдонима») для which:

```
$ \which vim dir which
/usr/bin/vim
/bin/dir
/usr/bin/which
```

Следующий пример, который работает только на некоторых системах Linux, похож на предыдущий, но в нем используется псевдоним для which (который выводится самой утилитой):

```
$ which vim dir which
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot
        --show-tilde'
```

```
        /usr/bin/which
/usr/bin/vim
/usr/bin/dir
```

Последний пример аналогичен предыдущему, за исключением того, что он запускается из оболочки tcsh. Вместо утилиты which используется встроенная в tcsh команда which:

```
tcsh $ which vim dir which
/usr/bin/vim
/bin/dir
which: shell built-in command.      which: встроенная команда оболочки.
```

who

Вывод информации о пользователях, вошедших в систему.

who [ключи]
who am i

Утилита who выводит информацию о пользователях, вошедших в локальную систему. Эта информация включает имя каждого пользователя, название терминального устройства, время входа в систему и, если приемлемо, соответствующее имя удаленного хоста.

Аргументы

Когда утилите предоставлены два аргумента (традиционно это `am i`), who выводит информацию, касающуюся пользователя, запустившего команду. Применительно к ситуации, перед именем пользователя может быть выведено имя хоста той системы, с которой была запущена команда who (в виде `kudos!max`).

Ключи

При работе под управлением Linux утилита who допускает применение общих ключей, описание которых дано в табл. V.2. Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме специально оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`--all -a` Вывод всего объема информации. (Linux)

`--boot -b` Вывод даты и времени последней загрузки системы. (Linux)

`--heading -H` Вывод заголовка.

`--login -l` (буква «l» в нижнем регистре). Вывод списка устройств, ожидающих входа пользователя. (Linux)

`--count -q` (quick — быстрый). Вывод списка, состоящего только из имен пользователей, за которыми следует количество пользователей, вошедших в систему.

`--mesg -T` Добавление после каждого имени пользователя символа, показывающего, можно ли пользователю с этим именем послать сообщение. Плюс (+) означает,

что сообщения разрешены, дефис (-) означает, что они запрещены, а вопросительный знак (?) показывает, что утилита who не может найти устройство. Если сообщения разрешены, то для связи с пользователем можно воспользоваться утилитой write. См. раздел «mesg: Отказ в приеме или прием сообщений» на с. 92.

`--users -u` Включение в выводимую информацию времени бездействия каждого пользователя. Если пользователь что-нибудь набирал на своем терминале в последнюю минуту, в этом поле появляется точка (.). Если набора не было в течение более чем одного дня, появляется слово `old`. В качестве дополнения этот ключ включает поля PID-номера и комментария. См. подраздел «Комментарии».

Комментарии

Строка, выводимая утилитой who, имеет следующий синтаксис:

пользователь [сообщения] линия время_входа [простой] [PID] комментарий

Пользователь представляет собой имя пользователя. *Сообщение* показывает, разрешены или нет сообщения (см. описание ключа `-T` (`--mesg`)). *Линия* представляет имя устройства, связанного с той линией, по которой пользователь вошел в систему. *Время_входа* показывает дату и время входа пользователя в систему. *Простой* показывает продолжительность паузы, возникшей с момента последнего использования терминала (время бездействия, см. описание ключа `-u` (`--users`)). *PID* показывает идентификационный номер процесса. *Комментарий* показывает имя удаленной системы, с которой пользователь вошел в систему (для локальных пользователей это поле оставляется пустым).

Примечания

Утилита finger (см. раздел «finger: Вывод списка пользователей, вошедших в систему» на с. 88, а также описание этой утилиты в данной части книги) предоставляет информацию, аналогичную той, которая выдается утилитой who.

Примеры

В следующих примерах показано использование утилиты who:

```
$ who
hls      tty1     Jul 30 06:01
sam      tty2     Jul 30 06:02
max      ttyp3    Jul 30 14:56 (bravo)

$ who am i
bravo!max     ttyp3    Jul 30 14:56 (bravo)

$ who -HTu
USER        LINE     TIME         IDLE      PID COMMENT
hls       - tty1     Jul 30 06:01 03:53    1821
sam       + tty2     Jul 30 06:02 14:47    2235
max       + ttyp3    Jul 30 14:56   .     14777  (bravo)
```

xargs

Превращение стандартного ввода в командные строки.

xargs [ключи] [команда]

Утилита xargs предоставляет удобный и эффективный способ превращения стандартного вывода одной команды в аргументы для другой команды. Эта утилита осуществляет чтение данных со стандартного ввода, отслеживает максимальную разрешенную длину командной строки и не допускает превышения этого ограничения путем выдачи еще одной ***команды***, если в этом есть необходимость. И наконец, утилита xargs выполняет созданную командную строку (или строки).

Аргументы

Команда — это командная строка, которую нужно использовать утилите xargs в качестве основы для создаваемой ею команды. Если ***команда*** не указана, этот аргумент принимает по умолчанию значение echo. Утилита xargs добавляет к ***команде*** аргументы, получаемые со стандартного ввода. Если аргументам, получаемым со стандартного ввода, должны предшествовать какие-нибудь другие аргументы, их нужно включить в ***команду*** как ее часть.

Ключи

Ключи, перед которыми ставится двойной дефис (--), работают только под управлением Linux. Кроме специально оговоренных, однобуквенные ключи, перед которыми ставится одинарный дефис, работают как под управлением Linux, так и под управлением OS X.

`-I` [*маркер*] (replace — замена). Предоставление возможности помещать аргументы со стандартного ввода в любое место внутри ***команды***. Все появления *маркера* внутри ***команды*** для утилиты xargs заменяются аргументами, сгенерированными со стандартного ввода утилиты xargs. При использовании этого ключа xargs выполняет ***команду*** для каждой введенной строки. При использовании этого ключа игнорируется присутствие ключа `-l` (`--max-lines`).

`--max-lines` [`=`*количество*] `-l` [*количество*] (буква «l» в нижнем регистре). Однократное выполнение команды для каждой группы из *количества* строк ввода (по умолчанию *количество* равно 1). (Linux)

`--max-args=` *количество* `-n` *количество* Однократное выполнение команды для каждой группы, состоящей из *количества* аргументов в строке ввода.

`--max-procs=` *количество* `-P` *количество* Разрешение утилите xargs запускать одновременно до указанного *количества* экземпляров ***команды***. (По умолчанию используется аргумент 1, позволяющий запускать команды в последовательном режиме.) Этот ключ может повысить пропускную способность, если утилита xargs запущена на многопроцессорной системе.

`--interactive` `-p`	Вывод приглашения на подтверждение перед каждым выполнением команды.
`--no-run-if-empty` `-r`	Принуждение утилиты xargs к невыполнению ***команды***, если стандартный ввод пуст. Обычно xargs выполняет ***команду*** как минимум один раз, даже если стандартный ввод пуст. (Linux)

Комментарии

Утилита xargs читает аргументы для ***команды*** со стандартного ввода, интерпретируя каждое строковое значение, отделенное пробельными символами в качестве отдельного аргумента. Затем она создает командную строку из команды и ряда аргументов. Когда командная строка при добавлении еще одного аргумента достигнет максимально возможной длины, xargs запускает созданную командную строку. Если ввод продолжается, xargs повторяет процесс построения командной строки и ее запуска. Этот процесс продолжается до тех пор, пока не будет прочитан весь ввод.

Примечания

Утилита xargs часто используется в качестве эффективной альтернативы ключу `-exec` утилиты find (см. описание этой утилиты). При вызове find с ключом `-exec` для запуска команды эта утилита однократно запускает команду для каждого обрабатываемого с ее помощью файла. Каждое выполнение команды создает новый процесс, который может приводить к истощению системных ресурсов при обработке большого количества файлов. Путем аккумуляции наибольшего возможного количества аргументов утилита xargs может существенно сократить количество необходимых процессов. Первый пример в подразделе «Примеры» показывает порядок использования утилиты xargs с утилитой find.

Примеры

Для поиска и удаления из рабочего каталога и всех его подкаталогов всех файлов с именами, оканчивающимися на .o, можно воспользоваться утилитой find с ключом `-exec`:

```
$ find . -name \*.o -exec rm --force {} \;
```

При таком подходе утилита rm вызывается по одному разу для каждого файла с расширением .o, найденного утилитой find. Каждый вызов утилиты rm требует создания нового процесса. Если имеется большое количество файлов с расширением .o, система может потратить очень много времени на создание, запуск, а затем на ликвидацию последствий работы этих процессов. Количество процессов можно сократить, позволив утилите xargs перед вызовом rm собрать столько имен файлов, сколько позволит длина командной строки:

```
$ find . -name \*.o -print | xargs rm --force
```

В следующем примере в содержимом всех файлов *.txt, найденных утилитой find, ищутся строки, содержащие слово `login`. Все имена файлов, содержащие слово `login`, выводятся с помощью утилиты grep.

```
$ find . -name \*.txt -print | xargs grep -w -l login
```

В следующем примере показано, как нужно использовать ключ -I, чтобы заставить утилиту xargs *вместо добавления стандартного ввода к* ***команде*** вставить его в эту ***команду***. Этот ключ также заставляет ***команду*** выполняться при каждой встрече в стандартном вводе символа разделителя строк (NEWLINE), это поведение утилиты не отменяется и при использовании ключа -l (--max-lines).

```
$ cat names
Tom,
Dick,
and Harry

$ xargs echo "Hello," < names
Hello, Tom, Dick, and Harry

$ xargs -I xxx echo "Hello xxx. Join me for lunch?" < names
Hello Tom,. Join me for lunch?
Hello Dick,. Join me for lunch?
Hello and Harry. Join me for lunch?
```

В последнем примере используется такой же файл ввода, как и в предыдущих примерах, а также используются ключи -n (--max-args) и -l (--max-lines):

```
$ xargs -n 1 echo "Hi there" < names
Hi there Tom,
Hi there Dick,
Hi there and
Hi there Harry

$ xargs -l 2 echo "Hi there" < names
Hi there Tom, Dick,
Hi there and Harry
```

Дополнительные примеры использования утилиты xargs даны в описании утилиты ssh, в подразделе «Примеры».

Марк Собель

Linux. Администрирование и системное программирование

2-е издание

Перевел с английского Н. Вильчинский

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Литературный редактор	*К. Кноп*
Художественный редактор	*Л. Адуевская*
Корректор	*В. Листова*
Верстка	*Е. Егорова*

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 17.03.11. Формат 70x100/16. Усл. п. л. 70,950. Тираж 2000. Заказ 63.23.
Отпечатано по технологии CtP в ООО «Северо-Западный Печатный двор»,
188300, Ленинградская обл., г. Гатчина, ул. Железнодорожная, 45Б.