

O'REILLY®

Паттерны Kubernetes

шаблоны разработки
собственных
облачных приложений



Билджин Ибрам
Роланд Хасс



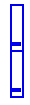
O'REILLY®

Паттерны Kubernetes

шаблоны разработки
собственных
облачных приложений



Билджин Ибрам
Роланд Хасс



Билджин Ибрам, Роланд Хасс
Паттерны Kubernetes: Шаблоны разработки собственных облачных приложений



2020

Научный редактор *М. Малявин*
Переводчик *А. Макарова*
Литературный редактор *А. Руденко*
Художники *В. Мостипан, А. Шляго (Шантурова)*
Корректоры *С. Беляева, Н. Викторова*
Верстка *Л. Егорова*

Билджин Ибрам, Роланд Хасс

Паттерны Kubernetes: Шаблоны разработки собственных облачных приложений. — СПб.: Питер, 2020.

ISBN 978-5-4461-1443-6

© [ООО Издательство "Питер"](#), 2020

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Предисловие

Начиная работу над фреймворком Kubernetes почти пять лет назад, мы — Крейг, Джо и я — понимали, что он способен изменить мир разработки и доставки программного обеспечения. Но мы не думали, что это изменение произойдет так быстро. В настоящее время Kubernetes служит основой для создания переносимых и надежных систем для основных общедоступных и частных облаков, а также не виртуализированных окружений. Несмотря на широкую распространенность фреймворка Kubernetes, благодаря которой кластер в облаке можно развернуть менее чем за пять минут, многие недостаточно четко представляют, что делать дальше, после создания этого кластера. Мы добились значительных успехов в практической реализации самого Kubernetes, но это только часть решения. Это фундамент, на котором создаются приложения. Он предлагает обширную библиотеку инструментов для их создания, но почти не дает советов и рекомендаций архитекторам или разработчикам приложений, как можно объединить различные части этого фундамента, чтобы получить законченную надежную систему, соответствующую целям и потребностям.

Представление о том, что же дальше делать с кластером Kubernetes, можно получить из прошлого опыта работы с аналогичными системами или прибегнуть к методу проб и ошибок, но такой путь обходится слишком дорого с точки зрения времени и качества систем, предлагаемых нашим конечным пользователям. Для тех, кто решит предоставлять критически важные услуги на основе таких систем, как Kubernetes, обретение опыта методом проб и ошибок займет слишком много времени и приведет к очень серьезным проблемам, связанным с простоями и сбоями.

Вот почему книга Билджина и Роланда имеет особую ценность. «Паттерны Kubernetes» знакомят вас с опытом, который мы вложили в API и инструменты, составляющие Kubernetes. Фреймворк Kubernetes является воплощением опыта, накопленного сообществом разработчиков, занимающихся созданием высоконадежных распределенных систем в разных окружениях. Каждый объект и каждая возможность, добавленные в Kubernetes, — это основополагающий инструмент, разработанный и созданный специально для удовлетворения конкретной потребности. В этой книге рассказывается, как использовать идеи, заложенные в Kubernetes, для решения практических задач и построения своей системы.

Работая над Kubernetes, мы всегда говорили, что наша главная цель — максимально упростить разработку распределенных систем, и именно такие книги наглядно показывают, насколько мы преуспели в этом. Билджин и Роланд отобрали основные инструменты разработчика Kubernetes и разбили их на группы, упростив их изучение и применение. К концу этой книги вы будете знать не только о компонентах, доступных вам в Kubernetes, но и о том, «как» и «зачем» строить системы с использованием этих компонентов.

Брендан Бернс (Brendan Burns), разработчик Kubernetes

Вступление

В последние годы с развитием микросервисов и контейнеров способы проектирования, разработки и эксплуатации ПО значительно изменились. Современные приложения оптимизируются в целях масштабируемости, эластичности, отказоустойчивости и быстрого изменения. Для соответствия новым принципам эти современные архитектуры требуют другого набора паттернов и практик. Цель этой книги — помочь разработчикам создавать облачные приложения с использованием Kubernetes в качестве платформы времени выполнения. Для начала кратко познакомимся с двумя основными составляющими этой книги: фреймворком Kubernetes и паттернами проектирования.

Kubernetes

Kubernetes — это платформа для управления контейнерами. Зарождение Kubernetes произошло где-то в центрах обработки данных компании Google, где появилась внутренняя платформа управления контейнерами Borg (<https://research.google.com/pubs/pub43438.html>). Платформа Borg много лет использовалась в Google для запуска приложений. В 2014 году Google решил передать свой опыт работы с Borg новому проекту с открытым исходным кодом под названием *Kubernetes* (в переводе с греческого «кормчий», «рулевой»), а в 2015 году он стал первым проектом, переданным в дар недавно основанному фонду Cloud Native Computing Foundation (CNCF).

С самого начала проект Kubernetes приобрел целое сообщество пользователей, и число участников росло

невероятно быстрыми темпами. В настоящее время Kubernetes считается одним из самых активных проектов на GitHub. Можно даже утверждать, что на момент написания этой книги Kubernetes был наиболее часто используемой и многофункциональной платформой управления контейнерами. Kubernetes также формирует основу других платформ, построенных поверх него. Наиболее известной из таких систем вида «платформа как услуга» (Platform-as-a-Service) является Red Hat OpenShift, которая добавляет в Kubernetes различные дополнительные возможности, в том числе способы создания приложений на этой платформе. Это только часть причин, по которым мы выбрали Kubernetes в качестве эталонной платформы для описания паттернов использования облачных систем в этой книге.

Эта книга предполагает наличие у читателя некоторых базовых знаний о Kubernetes. В главе 1 мы перечислим основные понятия Kubernetes и заложим основу для обсуждения паттернов в следующих главах.

Паттерны проектирования

Понятие *паттернов*, или *шаблонов*, *проектирования* появилось в 1970-х годах в области архитектуры. Кристофер Александер (Christopher Alexander), архитектор и системный теоретик, и его команда опубликовали в 1977 году новаторский труд «A Pattern Language»¹ (Oxford University Press), в котором описываются архитектурные шаблоны создания городов, зданий и других строительных проектов. Некоторое время спустя эта идея была принята недавно сформировавшейся индустрией программного обеспечения. Самая известная книга в этой области — «Приемы объектно-ориентированного проектирования. Паттерны проектирования» Эриха Гаммы,

Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса — «Банды четырех» (Addison-Wesley). Когда мы говорим об известных паттернах «Одиночка» (Singleton), «Фабрика» (Factories) или «Делегирование» (Delegation), то используем названия, данные в этой книге. С тех пор было написано много других замечательных книг о паттернах для различных областей с разной степенью подробностей, таких как «Enterprise Integration Patterns»² Грегора Хопа (Gregor Hohpe) и Бобби Вульфа (Bobby Woolf) или «Patterns of Enterprise Application Architecture»³ Мартина Фаулера (Martin Fowler).

Проще говоря, паттерн описывает *повторимое решение задачи*⁴. Паттерн отличается от рецепта тем, что вместо пошаговых инструкций для решения задачи он определяет план решения целого класса подобных задач. Например, паттерн Александра «Пивная» описывает, как следует строить питейные заведения, чтобы они стали местами, где «незнакомцы и друзья становятся собутыльниками», а не «пристанищами для одиночек». Все заведения, построенные по этому шаблону, выглядят по-разному, но имеют общие черты, такие как открытые ниши для групп от четырех до восьми человек и общий зал, где сотни людей могут вместе выпивать, веселиться, слушать музыку или делать что-то еще.

Однако паттерны не просто предоставляют решения. Они также формируют язык. Уникальные названия паттернов образуют компактный язык, в основе которого лежат существительные, и каждый паттерн имеет уникальное *название*. Когда люди упоминают эти названия в разговорах между собой, они автоматически вызывают у них похожие ментальные представления. Например, когда мы говорим о столе, любой, кто слышит нас, предполагает, что мы говорим о деревянной столешнице на четырех ножках, на которую можно класть разные вещи. То же происходит в программной

инженерии, когда мы говорим о «фабрике». В контексте объектно-ориентированного программирования мы немедленно связываем с термином «фабрика» некий объект, который производит другие объекты. Поскольку мы уже знаем решение, лежащее в основе паттерна, то можем перейти к решению еще не решенных проблем.

Есть и другие характеристики языка паттернов. Паттерны взаимосвязаны между собой и могут перекрываться, поэтому вместе охватывают большую часть пространства задач. Кроме того, как отмечается в книге «Язык паттернов», паттерны имеют разные уровни детализации и области действия. Более общие паттерны охватывают более широкий спектр задач и предлагают приблизительные рекомендации, касающиеся их решения. Специализированные паттерны дают очень конкретное решение, но применяются не так широко. Эта книга содержит все виды паттернов, и многие паттерны ссылаются на другие паттерны или даже могут включать другие паттерны как часть решения.

Другая особенность паттернов заключается в том, что они следуют жесткому формату. Однако каждый автор определяет свой формат, и, к сожалению, нет единого стандарта, который определял бы, как должны излагаться паттерны. Мартин Фаулер дает превосходный обзор форматов, используемых для языков паттернов, в своей статье «Writing Software Patterns» (<http://bit.ly/2HluUdJ>).

Структура книги

Мы выбрали простой паттерн структуры книги. Мы не придерживаемся какого-либо конкретного языка и для описания каждого паттерна используем следующую структуру:

Название

Каждый паттерн имеет название, которое также является названием главы. Названия образуют основу языка паттернов.

Задача

В этом разделе дается широкий контекст и подробное описание пространства паттерна.

Решение

Этот раздел рассказывает, как паттерн решает проблему способом, характерным для Kubernetes. Этот раздел также содержит ссылки на другие паттерны, которые либо связаны, либо являются частью данного паттерна.

Пояснение

Обзор достоинств и недостатков решения в данном контексте.

Дополнительная информация

Этот заключительный раздел содержит источники дополнительной информации, касающейся паттерна.

Мы организовали паттерны в этой книге следующим образом:

- *Часть I Основные паттерны* охватывает основные понятия Kubernetes и перечисляет основополагающие принципы и

практики создания облачных приложений на основе контейнеров.

- *Часть II Поведенческие паттерны* описывает паттерны, основанные на базовых паттернах, и добавляют специализированные идеи управления взаимодействиями между контейнерами и платформой.
- *Часть III Структурные паттерны* содержит паттерны, имеющие отношение к организации контейнеров в *поды* (pod) — элементарные единицы платформы Kubernetes.
- *Часть IV Конфигурационные паттерны* дает представление о различных способах настройки приложений в Kubernetes. Это очень детализированные паттерны, включающие конкретные рецепты для подключения приложений к их конфигурациям.
- *Часть V Дополнительные паттерны* знакомит с дополнительными понятиями, например, как можно расширить саму платформу или как создавать образы контейнеров непосредственно внутри кластера.

Паттерны не всегда вписываются в какую-то одну категорию. В зависимости от контекста один и тот же паттерн может вписываться в несколько категорий. Каждая глава посвящена одному паттерну и является независимой, поэтому вы можете читать главы по отдельности и в любом порядке.

Кому адресована эта книга

Эта книга адресована *разработчикам*, которые хотят проектировать и разрабатывать облачные приложения для

платформы Kubernetes. Наибольшую пользу из нее извлекут читатели, которые хотя бы немного знакомы с контейнерами и понятиями Kubernetes и хотят подняться на новый уровень. Однако вам не нужно знать низкоуровневые детали устройства Kubernetes, чтобы понять варианты и паттерны использования. Архитекторы, технические консультанты и разработчики выиграют от знакомства с паттернами, описанными здесь.

Эта книга основана на сценариях использования и уроках, извлеченных из реальных проектов. Мы хотим помочь вам создавать облачные приложения, а не изобретать велосипед.

Что вы узнаете

Вы сделаете массу открытий. Некоторые паттерны могут выглядеть как выдержки из руководства по Kubernetes, но при ближайшем рассмотрении вы увидите, что паттерны представлены с концептуальной точки зрения, чего не хватает в других книгах, посвященных этой теме. Другие описываются иначе, с подробными рекомендациями для каждой конкретной задачи, как, например, в части IV *Конфигурационные паттерны*.

Независимо от степени подробности описания паттерна, вы узнаете все, что Kubernetes предлагает для каждого конкретного паттерна, со множеством иллюстративных примеров. Все эти примеры были протестированы, и мы расскажем, как получить их исходный код в разделе «Использование примеров кода».

Прежде чем начать погружение, кратко перечислим, чем не является эта книга:

- Эта книга не является руководством по настройке самого кластера Kubernetes. Каждый паттерн и каждый пример предполагает, что вы уже настроили и запустили Kubernetes.

Опробовать примеры можно несколькими способами. Желая узнать, как настроить кластер Kubernetes, рекомендуем книгу «Managing Kubernetes» Брендана Бернса (Brendan Burns) и Крейга Трейси (Craig Tracey), изданную в O'Reilly (<https://oreil.ly/2HoadnU>). Кроме того, книга «Kubernetes Cookbook» Майкла Хаузенбласа (Michael Hausenblas) и Себастьяна Гоасгена (Sébastien Goasguen), изданная в O'Reilly (<http://bit.ly/2FTgJzk>), содержит рецепты создания кластера Kubernetes с нуля.

- Эта книга не является ни введением в Kubernetes, ни справочным руководством. Мы затрагиваем многие особенности Kubernetes и объясняем их до определенной степени, но основное внимание уделяется идеям, лежащим в основе этих особенностей. В главе 1 «Введение» предлагается краткий обзор основ Kubernetes. Если вы ищете исчерпывающую книгу об использовании Kubernetes, мы настоятельно рекомендуем книгу «Kubernetes in Action»⁵ Марко Лукши (Marko Lukša), изданную в Manning Publications.

Книга написана в непринужденной манере и больше напоминает серию очерков, которые можно читать независимо.

Типографские соглашения

Как уже упоминалось, паттерны образуют простой, взаимосвязанный язык. Чтобы подчеркнуть эту взаимосвязь, названия паттернов записываются *курсивом* (например, *Sidecar* (*Прицеп*)). Когда паттерн получает название по базовому понятию Kubernetes (например, *Init Container*

(Инициализирующий контейнер) или *Controller* (Контроллер)), мы используем такой способ оформления только для прямых ссылок на сам паттерн. Там, где это имеет смысл, мы также даем ссылки на главы с описаниями паттернов для упрощения навигации.

Мы также используем следующие соглашения:

- Все, что вводится в командной оболочке или в редакторе, будет оформляться моноширинным шрифтом.
- Имена ресурсов Kubernetes всегда записываются с заглавной буквы (например, Pod). Если ресурс имеет комбинированное имя, такое как ConfigMap, мы используем его вместо более естественного обозначения «config map» (конфигурационная карта), чтобы подчеркнуть, что имеется в виду понятие Kubernetes.
- Имена некоторых ресурсов Kubernetes совпадают с общими понятиями, такими как «служба» или «узел». В этих случаях мы используем оформление, характерное для имен ресурсов, только для ссылок на ресурсы.

Использование примеров кода

Описание каждого паттерна сопровождается примерами, которые вы можете найти на веб-странице книги (<https://k8spatterns.io/>). Также ссылки на примеры для каждого паттерна приводятся в разделе «Дополнительная информация» в каждой главе.

В разделе «Дополнительная информация» также приводятся ссылки на дополнительную информацию, имеющую отношение к паттерну. Мы постоянно обновляем эти списки в

репозитории примеров. Изменения в коллекциях ссылок также будут публиковаться в Twitter (<https://twitter.com/k8spatterns>).

Исходный код всех примеров в этой книге доступен в GitHub (<https://github.com/k8spatterns>). Репозиторий и веб-сайт также содержат указания и инструкции о том, как создать кластер Kubernetes для опробования примеров. Просматривая примеры, также загляните в предоставляемые файлы ресурсов. В них вы найдете много полезных комментариев, помогающих понять код примера.

Во многих примерах используется REST-служба *random-generator*, которая возвращает случайные числа. Она специально создавалась для опробования примеров из этой книги. Исходный код этой службы тоже можно найти в GitHub (<https://github.com/k8spatterns/random-generator>), а ее образ для развертывания в контейнере `k8spatterns/random-generator` добавлен в каталог Docker Hub.

Для описания полей ресурсов мы используем обозначение путей в формате JSON. Например, `.spec.replicas` — это ссылка на поле `replicas` в разделе `spec` ресурса.

Если вы найдете ошибку в коде примера или в документации или если у вас появится вопрос, смело создавайте заявку в трекере проблем GitHub (<https://github.com/k8spatterns/examples/issues>). Мы следим за появлением заявок и с радостью отвечаем на любые вопросы.

Мы приветствуем любые предложения по изменению кода! Если вы считаете, что сможете усовершенствовать примеры, мы будем рады рассмотреть ваши предложения. Просто создайте заявку или запрос на трекере проблем GitHub и начните диалог с нами.

Благодарности

Создание этой книги было долгим путешествием, продолжавшимся два года, и мы хотим поблагодарить всех наших рецензентов, помогавшим нам не сбиться с пути. Особую благодарность хотим выразить Паоло Антинори (Paolo Antinori) и Андреа Тарокки (Andrea Tarocchi), помогавшим нам в этом путешествии. Большое спасибо также Марко Лукше (Marko Lukša), Брендону Филипсу (Brandon Philips), Майклу Хуттерманну (Michael Hüttermann), Брайану Грейсли (Brian Gracely), Эндрю Блоку (Andrew Block), Иржи Кремсеру (Jiri Kremser), Тобиасу Шнеку (Tobias Schneck) и Рикку Вагнеру (Rick Wagner), которые поддержали нас своим опытом и советами. Наконец, но не в последнюю очередь, большое спасибо нашим редакторам Вирджинии Уилсон (Virginia Wilson), Джону Девинсу (John Devins), Кэтрин Тозер (Katherine Tozer), Кристине Эдвардс (Christina Edwards) и всем замечательным сотрудникам O'Reilly за то, что помогли довести эту книгу до финала.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу **comp@piter.com** (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства **www.piter.com** вы найдете подробную информацию о наших книгах.

¹ *Кристофер Александер, Сара Исикава, Мюррей Силверстайн. Язык шаблонов. Города. Здания. Строительство. Издательство Студии Артемия Лебедева, 2014. — Примеч. пер.*

² *Хоп Грегор, Вульф Бобби. Паттерны интеграции корпоративных приложений. М.: Вильямс, 2016. — Примеч. пер.*

[3](#) *Мартин Фаулер, Дейвид Райс, Мэттью Фоммел, Эдвард Хайет, Роберт Ми, Рэнди Стаффорд. Паттерны корпоративных приложений. М.: Вильямс, 2016. — Примеч. пер.*

[4](#) Кристофер Александер и его команда определили первоначальное значение слова «паттерн» в контексте архитектуры следующим образом: «Каждый паттерн дает описание той или иной проблемы, регулярно возникающей в окружающем нас пространстве, вслед за которым представлена суть решения данной проблемы, сформулированная таким образом, чтобы вы могли многократно использовать это решение, но никогда не копировать его» (*Кристофер Александер, Сара Исикава, Мюррей Силверстайн. Язык шаблонов. Города. Здания. Строительство. С. 20*). Мы считаем, что это определение прекрасно подходит также для паттернов, которые описываются в этой книге, с той лишь разницей, что у нас, пожалуй, не так много свободы в реализации решений.

[5](#) *Лукиа Марко. Kubernetes в действии. М.: ДМК Пресс, 2018. — Примеч. пер.*

Глава 1. Введение

В этой вводной главе мы подготовим основу для остальной части книги и обсудим важные понятия Kubernetes, используемые в проектировании и реализации облачных приложений на основе контейнеров. Понимание этих новых абстракций, а также связанных с ними принципов и паттернов из этой книги является ключом к созданию распределенных приложений, автоматизируемых облачными платформами.

Эта глава не является обязательным условием для понимания паттернов, описываемых далее. Читатели, знакомые с понятиями Kubernetes, могут пропустить ее и сразу перейти к интересующей их категории.

Путь в облачное окружение

Наибольшей популярностью среди архитектур приложений для облачных платформ, таких как Kubernetes, пользуется архитектура микросервисов. Этот способ организации программного обеспечения помогает снизить сложность его разработки за счет дробления бизнес-функций и замены сложности разработки сложностью эксплуатации.

Существует большое количество теоретических и практических методов создания микросервисов с нуля или деления монолитных приложений на микросервисы. Большинство из этих методов основаны на приемах, описанных в книге Эрика Эванса (Eric Evans) «Domain-Driven Design»⁶ (Addison-Wesley), и понятиях ограниченного контекста и агрегатов. *Ограниченные контексты* непосредственно связаны с большими моделями и делят их на разные компоненты, и *агрегаты* помогают группировать

ограниченные контексты в модули с определенными границами транзакций. Однако кроме этих понятий, характерных для каждой предметной области, для каждой распределенной системы, независимо от того, основана она на микросервисах или нет, существует множество технических проблем, связанных с их организацией, структурой и поведением во время выполнения.

Контейнеры и механизмы управления контейнерами, такие как Kubernetes, предлагают много новых примитивов и абстракций для решения проблем распределенных приложений, и здесь мы обсудим разные варианты, которые следует учитывать при переносе распределенной системы в Kubernetes.

В этой книге мы будем исследовать особенности взаимодействий контейнеров и платформ, рассматривая контейнеры как черные ящики. Однако мы включили этот раздел, чтобы подчеркнуть важность внутреннего устройства контейнеров. Контейнеры и облачные платформы дают огромные преимущества распределенным приложениям, но, поместив мусор в контейнеры, вы получите распределенный мусор в большем масштабе. На рис. 1.1 показано, какие навыки необходимы для создания хороших облачных приложений.

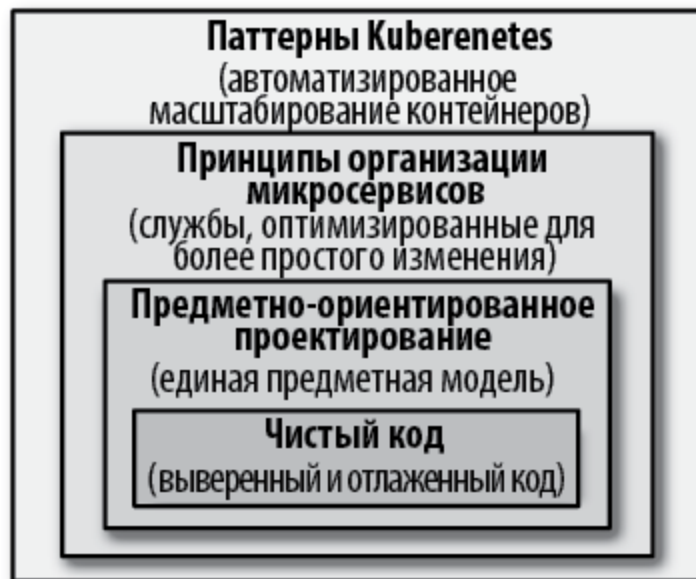


Рис. 1.1. Путь в облачное окружение

В общем случае каждое облачное приложение имеет несколько уровней абстракции, которые требуют различных проектных решений:

- В самом низу находится *уровень программного кода*. На этом уровне каждая переменная, каждый метод и каждый класс, которые вы создаете, оказывают прямое влияние на обслуживание приложения в долгосрочной перспективе. Независимо от технологии контейнеров и платформы управления ими, команда разработчиков и создаваемые ими артефакты будут иметь наибольшее влияние. Важно поддерживать разработчиков, которые стремятся писать чистый код, имеют необходимое количество автоматических тестов, постоянно улучшают качество кода и являются мастерами в сфере разработки программного обеспечения.
- *Предметно-ориентированное проектирование* (Domain-Driven Design, DDD) — это подход к проектированию программного

обеспечения с позиции бизнеса с целью получить архитектуру, как можно более близкую к реальному миру. Он лучше всего соответствует объектно-ориентированным языкам программирования, но есть и другие хорошие подходы к моделированию и проектированию программного обеспечения для решения практических задач. Модель с правильно выбранными границами, простыми в использовании интерфейсами и многофункциональным API является основой для успешной контейнеризации и автоматизации в дальнейшем.

- *Архитектурный стиль микросервисов* очень быстро стал нормой и определяет ценные принципы и методы проектирования часто изменяющихся распределенных приложений. Применение этих принципов позволяет создавать реализации, оптимизированные для масштабирования, отказоустойчивости и частых изменений, что является общим требованием для любого современного программного обеспечения.
- *Контейнеры* очень быстро превратились в стандартный способ упаковки и запуска распределенных приложений. Создание модульных, многоразовых контейнеров, которые прекрасно подходят для использования в облачных окружениях, является еще одной фундаментальной предпосылкой. С ростом числа контейнеров в каждой организации возникает необходимость управлять ими, используя более эффективные методы и инструменты. *Облачный* — это относительно новый термин, используемый для описания принципов, паттернов и инструментов автоматизации масштабирования контейнерных микросервисов. Мы будем использовать как взаимозаменяемые слова *облачный* и *Kubernetes*, последнее

из которых является названием наиболее популярной в настоящее время облачной платформы с открытым исходным кодом.

В этой книге мы не рассматриваем приемы разработки чистого кода, предметно-ориентированного проектирования и создания микросервисов. Все внимание мы сосредоточим исключительно на шаблонах и методах решения задач управления контейнерами. Но чтобы эти паттерны были максимально эффективными, приложение само должно быть тщательно спроектировано с применением методов разработки чистого кода, предметно-ориентированного проектирования, создания микросервисов и других соответствующих методик.

Распределенные примитивы

Чтобы объяснить, что подразумевается под новыми абстракциями и примитивами, мы сравним их с хорошо известным объектно-ориентированным программированием (ООП), например, на языке Java. Во вселенной ООП используются такие понятия, как класс, объект, пакет, наследование, инкапсуляция и полиморфизм. Среда выполнения Java предоставляет конкретные функции и гарантии управления жизненным циклом наших объектов и приложения в целом.

Язык Java и виртуальная машина Java (Java Virtual Machine, JVM) предоставляют локальные, внутрипроцессные строительные блоки для создания приложений. Kubernetes добавляет в эту привычную картину совершенно новое измерение, предлагая новый набор распределенных примитивов и среды выполнения для создания

распределенных систем, разбросанных по нескольким узлам и процессам. Используя Kubernetes для реализации всего поведения приложения, мы больше не полагаемся только на локальные примитивы.

Таблица 1.1. Локальные и распределенные примитивы²

Понятие	Локальный примитив	Распределенный примитив
Инкапсуляция поведения	Класс	Образ контейнера
Экземпляр поведения	Объект	Контейнер
Единица повторного использования	<i>.jar</i>	Образ контейнера
Композиция	Класс А содержит класс В	Шаблон Sidecar (Прицеп)
Наследование	Класс А расширяет класс В	Контейнер А создается из родительского образа
Единица развертывания	<i>.jar/.war/.ear</i>	Под (Pod)
Изоляция времени сборки/выполнения	Модуль, Пакет, Класс	Пространство имен, под, контейнер
Начальная инициализация	Конструктор	Инициализирующие контейнеры, или Init-контейнеры
Операции, следующие сразу за начальной инициализацией	Метод Init	postStart
Операции, непосредственно предшествующие уничтожению экземпляра	Метод Destroy	preStop
Процедура освобождения ресурсов	<i>finalize()</i> , обработчик события завершения	Деинициализированный контейнер ¹
Асинхронное и параллельное выполнение	ThreadPoolExecutor, ForkJoinPool	Задание
Периодическое выполнение	Timer, ScheduledExecutorService	Планировщик заданий

Понятие	Локальный примитив	Распределенный примитив
Фоновое выполнение	Фоновые потоки выполнения	Контроллер набора демонов (DaemonSet)
Управление конфигурацией	<code>System.getenv()</code> , <code>Properties</code>	Карта конфигураций (ConfigMap), секрет (Secret)

Мы все еще должны использовать объектно-ориентированные строительные блоки для создания компонентов распределенного приложения, но дополнительно мы можем использовать примитивы Kubernetes для организации некоторых видов поведения приложения. В табл. 1.1 перечислены различные понятия из области разработки приложений и соответствующие им локальные и распределенные примитивы.

Внутрипроцессные и распределенные примитивы имеют общие черты, но их нельзя сравнивать непосредственно и они не являются взаимозаменяемыми. Они работают на разных уровнях абстракции и имеют разные предпосылки и гарантии. Некоторые примитивы должны использоваться вместе. Например, мы должны использовать классы для создания объектов и помещать их в образы контейнеров. Однако некоторые другие примитивы могут служить полноценной заменой поведения в Java, например, CronJob в Kubernetes может полностью заменить ExecutorService в Java.

А теперь рассмотрим несколько распределенных абстракций и примитивов из Kubernetes, которые особенно интересны для разработчиков приложений.

Контейнеры

Контейнеры — это строительные блоки для создания облачных приложений на основе Kubernetes. Проводя аналогию с ООП и

Java, образы контейнеров можно сравнить с классами, а контейнеры — с объектами. По аналогии с классами, которые можно расширять (наследовать) и таким способом изменять их поведение, мы можем создавать образы контейнеров, которые расширяют (наследуют) другие образы контейнеров, и таким способом изменять поведение. По аналогии с объектами, которые можно объединять и использовать их возможности, мы можем объединять контейнеры, помещая их в поды (Pod), и использовать результаты их взаимодействий.

Продолжая сравнение, можно сказать, что Kubernetes напоминает виртуальную машину Java, но разбросанную по нескольким хостам и отвечающую за запуск контейнеров и управление ими.

Init-контейнеры можно сравнить с конструкторами объектов; контроллеры DaemonSet похожи на потоки выполнения, действующие в фоновом режиме (как, например, сборщик мусора в Java). Поды можно считать аналогами контекста инверсии управления (Inversion of Control, IoC), используемого, например, в Spring Framework, где несколько объектов имеют общий управляемый жизненный цикл и могут напрямую обращаться друг к другу.

Параллели можно было бы проводить и дальше, но не глубоко. Однако следует отметить, что контейнеры играют основополагающую роль в Kubernetes, а создание модульных, многоразовых, специализированных образов контейнеров является основой успеха любого проекта и экосистемы контейнеров в целом. Но что еще можно сказать о контейнерах и их назначении в контексте распределенного приложения, помимо перечисления технических характеристик образов контейнеров, которые обеспечивают упаковку и изоляцию? Вот несколько основных особенностей контейнеров:

- Образ контейнера — это функциональная единица, решающая одну определенную задачу.
- Образ контейнера принадлежит одной команде и имеет свой цикл выпуска новых версий.
- Образ контейнера является самодостаточным — он определяет и несет в себе зависимости времени выполнения.
- Образ контейнера является неизменным: после создания он не изменяется, но может настраиваться.
- Образ контейнера имеет определенные зависимости времени выполнения и требования к ресурсам.
- Образ контейнера экспортирует четко определенные API для доступа к его возможностям.
- Контейнер обычно выполняется как один процесс Unix.
- Контейнер позволяет безопасно масштабировать его вверх и вниз в любой момент.

Кроме всех этих характеристик, правильный образ контейнера должен иметь модульную организацию, поддерживать параметризацию и многократное использование в разных окружениях, где он будет работать. Также он должен предусматривать параметризацию для разных вариантов использования. Наличие небольших, модульных и многократно используемых образов контейнеров помогает создавать более специализированные и надежные образы подобно большой библиотеке в мире языков программирования.

Поды

Рассматривая характеристики контейнеров, легко заметить, что они идеально подходят для реализации принципов микросервисов. Образ контейнера предоставляет единую функциональную единицу, принадлежит одной команде, имеет независимый цикл выпуска новых версий и обеспечивает развертывание и изоляцию среды времени выполнения. В большинстве случаев один микросервис соответствует одному образу контейнера.

Однако многие облачные платформы предлагают еще один примитив для управления жизненным циклом группы контейнеров, в Kubernetes он называется подом. *Под (Pod⁸)* — это атомарная единица планирования, развертывания и изоляции среды времени выполнения для группы контейнеров. Все контейнеры, входящие в состав одной группы, всегда планируются для выполнения на одном хосте, развертываются вместе и могут совместно использовать пространства имен файловой системы, сети и процесса. Подчинение единому жизненному циклу позволяет контейнерам в поде взаимодействовать друг с другом через файловую систему или через сеть с помощью локальных механизмов межпроцессных взаимодействий, если это необходимо (например, по соображениям производительности).

Как показано на рис. 1.2, на этапах разработки и сборки микросервис соответствует образу контейнера, который разрабатывается и выпускается одной группой. Но во время выполнения аналогом микросервиса является под, представляющий единицу развертывания, размещения и масштабирования. Единственный способ запустить контейнер — в ходе масштабирования или миграции — использовать абстракцию пода. Иногда под содержит несколько

контейнеров, например, когда контейнер с микросервисом использует вспомогательный контейнер во время выполнения, как показано в главе 15 «Шаблон Sidecar».

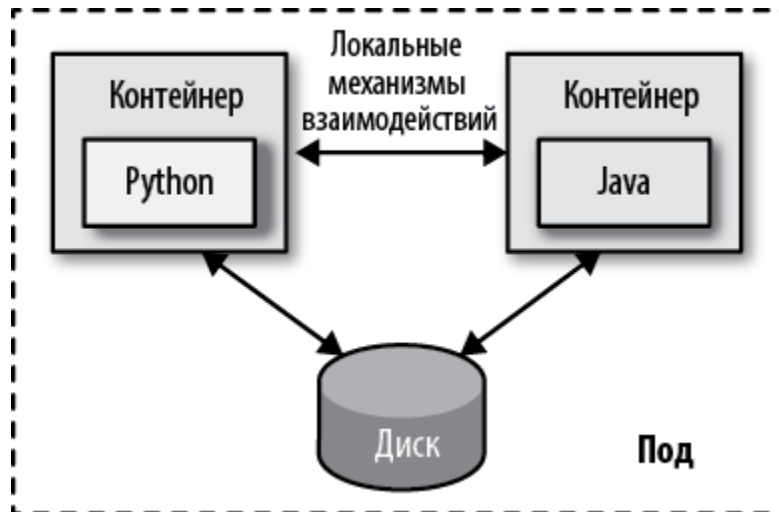


Рис. 1.2. Под как единица развертывания и управления

Уникальные характеристики контейнеров и подов образуют новый набор шаблонов и принципов разработки приложений на основе микросервисов. Мы рассмотрели некоторые характеристики хорошо спроектированных контейнеров; теперь рассмотрим некоторые характеристики пода:

- Под — это атомарная единица планирования. Собираясь запустить под, планировщик пытается найти хост, который удовлетворяет требованиям всех контейнеров, входящих в эту группу (есть некоторые особенности в отношении Init-контейнеров, которые мы рассмотрим в главе 14 «Init-контейнеры»). Если создать под со множеством контейнеров, планировщику придется отыскать хост, обладающий достаточными ресурсами для удовлетворения суммарных требований всех контейнеров. Процесс планирования описан в главе 6 «Автоматическое размещение».

- Под гарантирует совместное размещение контейнеров. Благодаря этому контейнеры в одной группе получают дополнительные возможности для взаимодействия друг с другом, из которых чаще всего используются общая локальная файловая система, сетевой интерфейс localhost и локальные механизмы межпроцессных взаимодействий (IPC).
- Под имеет IP-адрес, имя и диапазон портов, общих для всех контейнеров, входящих в группу. Это означает, что контейнеры в одной группе необходимо тщательно настраивать, чтобы избежать конфликтов портов, точно так же как параллельно выполняющиеся процессы Unix должны соблюдать осторожность при совместном использовании сетевого пространства хоста.

Под — это атом Kubernetes, в котором находится ваше приложение, но у вас нет прямого доступа к поду. Для этой цели используются службы.

Службы

Группы контейнеров, или поды, — это эфемерные образования, они могут появляться и исчезать в любое время по разным причинам: например, в ходе масштабирования вверх или вниз, в случае неудачи при проверке работоспособности контейнеров и при миграции узлов. IP-адрес группы становится известен только после того, как она будет запланирована и запущена на узле. Группу можно повторно запланировать для запуска на другом узле, если узел, на котором она выполнялась, прекратил работу. Все это означает, что сетевой адрес группы контейнеров может меняться в течение жизни приложения, и потому необходим

какой-то другой примитив для обнаружения и балансировки нагрузки.

Роль этого примитива играют службы (Services) Kubernetes. Служба — это еще одна простая, но мощная абстракция Kubernetes, которая присваивает имени службы постоянные IP-адрес и номер порта. То есть служба — это именованная точка входа для доступа к приложению. В наиболее распространенном сценарии служба играет роль точки входа для набора групп контейнеров, но это не всегда так. Служба — это универсальный примитив и может также служить точкой входа для доступа к функциональным возможностям за пределами кластера Kubernetes. Соответственно, примитив службы можно использовать для обнаружения служб и распределения нагрузки и для замены реализации и масштабирования без влияния на потребителей службы. Подробнее о службах мы поговорим в главе 12 «Обнаружение служб».

Метки

Как было показано выше, на этапе сборки аналогом микросервиса является контейнер, а на этапе выполнения — группа контейнеров. А что можно считать аналогом приложения, состоящего из нескольких микросервисов? Kubernetes предлагает еще два примитива, помогающих провести аналогию с понятием приложения: метки и пространства имен. Мы подробно рассмотрим пространства имен в разделе «Пространства имен» ниже.

До появления микросервисов понятию приложения соответствовала одна единица развертывания с единой схемой управления версиями и циклом выпуска новых версий. Приложение помещалось в один файл *.war*, *.ear* или в каком-то другом формате. Но затем приложения были разделены на

микросервисы, которые разрабатываются, выпускаются, запускаются, перезапускаются и масштабируются независимо друг от друга. С появлением микросервисов понятие приложения стало более размытым — больше нет ключевых артефактов или действий, которые должны выполняться на уровне приложения. Однако если понадобится указать, что некоторые независимые службы принадлежат одному приложению, можно использовать *метки*. Давайте представим, что одно монолитное приложение мы разделили на три микросервиса, а другое — на два.

В этом случае мы получаем пять определений групп контейнеров (и, может быть, много экземпляров этих групп), которые не зависят друг от друга с точки зрения разработки и времени выполнения. Тем не менее нам все еще может потребоваться указать, что первые три пода представляют одно приложение, а два других — другое приложение. Поды могут быть независимыми и представлять определенную ценность для бизнеса по отдельности, а могут зависеть друг от друга. Например, один под может содержать контейнеры, отвечающие за интерфейс с пользователем, а два других — за реализацию основной функциональности. Если какой-то из подов прекратит работать, приложение окажется бесполезным с точки зрения бизнеса. Использование меток дает возможность определять набор подов и управлять им как одной логической единицей. На рис. 1.3 показано, как можно использовать метки для группировки частей распределенного приложения в конкретные подсистемы.

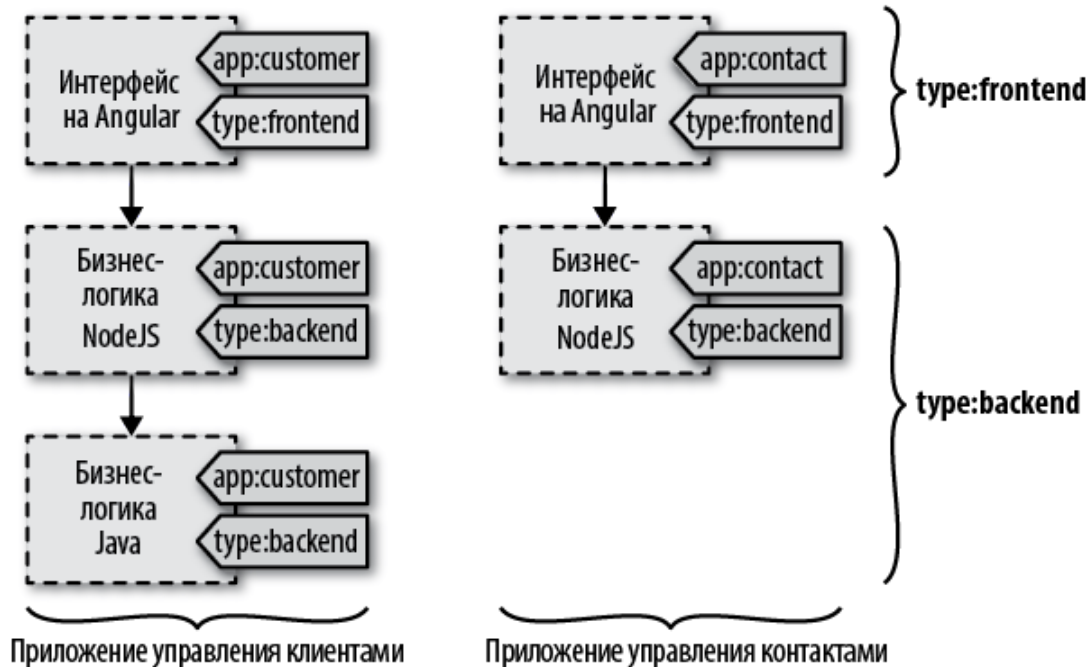


Рис. 1.3. Метки используются для идентификации подов как принадлежащих одному приложению

Вот несколько примеров использования меток:

- Метки используются наборами реплик (ReplicaSet) для поддержания в рабочем состоянии некоторых экземпляров подов. Для этого каждое определение пода должно иметь уникальную комбинацию меток, используемых для планирования.
- Метки также широко используются планировщиком для совместного размещения или распределения групп контейнеров по узлам с учетом требований этих групп.
- Метка может задавать логическую группировку набора подов и идентифицировать его как приложение.
- В дополнение к уже перечисленным типичным случаям метки можно использовать для хранения метаданных. Часто трудно заранее предусмотреть все случаи, когда могут

пригодиться метки, но лучше иметь достаточно меток, чтобы описать все важные аспекты подов. Например, могут пригодиться метки для описания логических групп внутри приложения, бизнес-характеристик и степени важности, конкретных зависимостей среды времени выполнения, таких как архитектура оборудования или настройки местоположения.

Впоследствии эти метки могут использоваться планировщиком для более точного планирования. Те же метки можно использовать из командной строки для управления соответствующими подами. Однако не следует выходить за рамки разумного и заранее добавлять слишком много меток. При необходимости требуемые метки всегда можно добавить позже. Удаление меток, кажущихся ненужными, намного опаснее, поскольку нет простого способа узнать, для чего они используются и какой эффект может вызвать их удаление.

Аннотации

Другой примитив — *аннотации* — очень похож на метки. Подобно меткам, аннотации организованы в виде ассоциативного массива, но предназначены для определения метаданных, которые используются компьютером, а не человеком.

Информация в аннотациях не предназначена для использования в запросах и сопоставления объектов. Аннотации предназначены для присоединения дополнительных метаданных к объектам, созданным разными инструментами и библиотеками. Например, аннотации можно использовать для добавления номера версии и сборки, информации об образе, временных меток, имен веток в

репозитории Git, номеров запросов на включение, хешей образов изображений, адресов в реестре, имен авторов, сведений об инструментах и многого другого. То есть метки используются главным образом для поиска и выполнения действий с соответствующими ресурсами, а аннотации — для прикрепления метаданных, которые могут использоваться компьютером.

Пространства имен

Другой примитив, который также может помочь в управлении группой ресурсов, — *пространство имен* Kubernetes. Как уже отмечалось выше, пространство имен может показаться похожим на метку, но в действительности это совсем другой примитив со своими характеристиками и назначением.

Пространства имен Kubernetes позволяют разделить кластер Kubernetes (который обычно распределяется по нескольким хостам) на логические пулы ресурсов. Пространства имен определяют области для ресурсов Kubernetes и предоставляют механизм для авторизации и применения других политик к сегменту кластера. Чаще всего пространства имен используются для организации разных программных окружений, таких как окружения для разработки, тестирования, интеграционного тестирования или промышленной эксплуатации. Пространства имен также можно использовать для организации многопользовательских архитектур (multitenancy), а также для изоляции рабочих групп, проектов и даже конкретных приложений. Но, в конечном счете, для полной изоляции некоторых окружений пространств имен недостаточно, поэтому создание отдельных кластеров является обычным явлением. Как правило, создается один непромышленный кластер Kubernetes, используемый, например, для разработки, тестирования и интеграционного

тестирования, и другой — промышленный кластер Kubernetes для опытной и промышленной эксплуатации.

Давайте перечислим некоторые особенности пространств имен и посмотрим, как они могут помочь нам в разных сценариях:

- Пространство имен управляется как ресурс Kubernetes.
- Пространство имен создает изолированную область для таких ресурсов, как контейнеры, группы контейнеров (поды), службы или наборы реплик (ReplicaSet). Имена ресурсов должны быть уникальными внутри пространства имен, но могут повторяться в разных пространствах имен.
- По умолчанию пространства имен определяют изолированную область для ресурсов, но ничто не изолирует эти ресурсы и не препятствует доступу одного ресурса к другому. Например, группа контейнеров из пространства имен для разработки может обращаться к другой группе контейнеров из пространства имен для промышленной эксплуатации, если ей известен IP-адрес этой группы. Однако существуют плагины для Kubernetes, которые обеспечивают изоляцию сети для достижения истинной многоарендности.
- Некоторые другие ресурсы, такие как сами пространства имен, узлы и постоянные тома (PersistentVolume), не принадлежат к пространствам имен и должны иметь уникальные имена для всего кластера.
- Каждая служба Kubernetes принадлежит к пространству имен и получает соответствующий DNS-адрес, включающий пространство имен в форме `<имя-службы>.<имя-пространства-имен>.svc.cluster.local`. То есть имя

пространства имен присутствует в URI каждой службы, принадлежащей к данному пространству имен. Это одна из причин, почему так важно выбирать правильные имена для пространств имен.

- Квоты ресурсов (ResourceQuota) определяют ограничения на совокупное потребление ресурсов пространством имен. С помощью квот ResourceQuota администратор кластера может управлять количеством объектов каждого типа в пространстве имен. Например, используя квоты, он может определить, что пространство имен для разработки может иметь только пять карт конфигураций (ConfigMap), пять объектов Secret, пять служб, пять наборов реплик, пять постоянных томов и десять подов.
- Квоты ресурсов (ResourceQuota) также могут ограничивать общую сумму вычислительных ресурсов, доступных для запроса в данном пространстве имен. Например, в кластере с 32 Гбайт ОЗУ и 16 ядрами можно выделить половину ресурсов — 16 Гбайт ОЗУ и 8 ядер — для промышленной эксплуатации, 8 Гбайт ОЗУ и 4 ядра для промежуточного окружения, 4 Гбайт ОЗУ и 2 ядра для разработки и столько же для тестирования. Возможность наложения ограничений на группы объектов с использованием пространств имен и квот ресурсов неоценима.

Пояснение

Мы кратко рассмотрели лишь часть основных понятий Kubernetes, которые используются в этой книге. Примитивов, которыми разработчики пользуются каждый день, намного больше. Например, создавая контейнерную службу, вы можете задействовать множество объектов Kubernetes, которые

позволяют воспользоваться всеми преимуществами фреймворка. Имейте в виду, что это только объекты, используемые разработчиками приложений для интеграции контейнерной службы в Kubernetes. Есть и другие понятия, используемые администраторами, чтобы позволить разработчикам эффективно управлять платформой. На рис. 1.4 представлены основные ресурсы Kubernetes, которые могут пригодиться разработчикам.

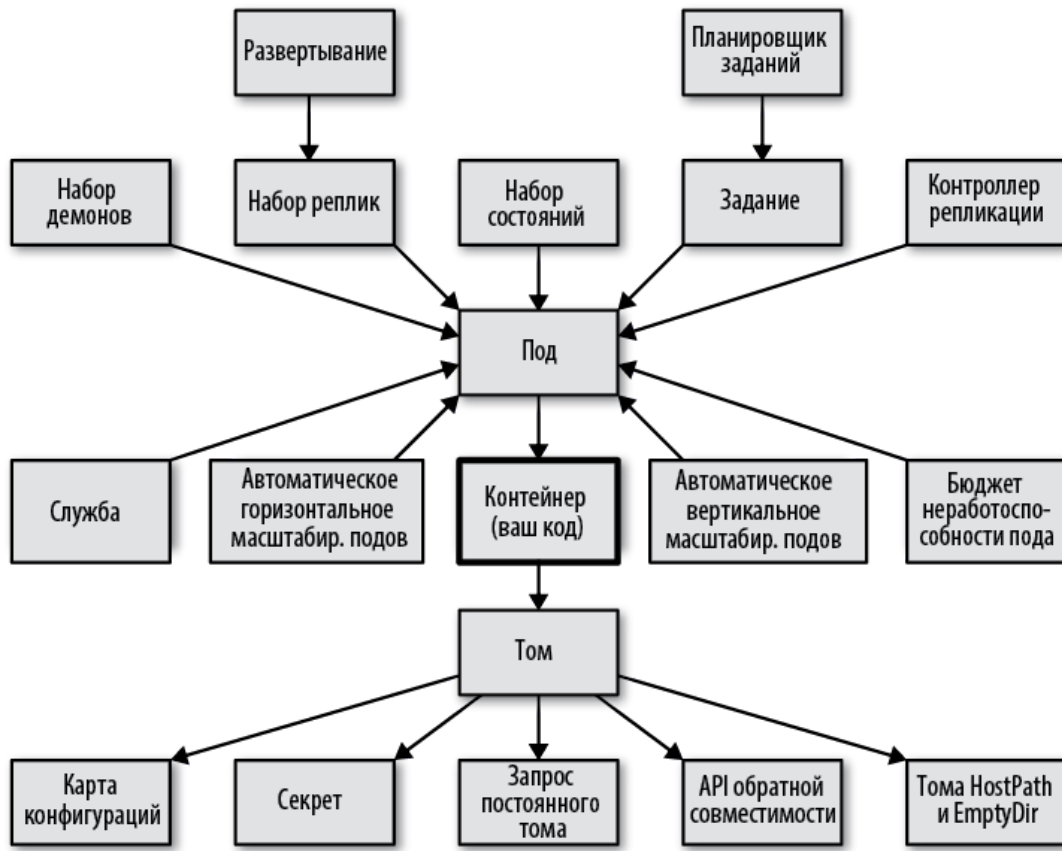


Рис. 1.4. Понятия Kubernetes для разработчиков

Эти новые примитивы способствуют появлению новых способов решения проблем, наиболее универсальные из которых становятся шаблонами. В этой книге мы не будем подробно описывать все решения, а сосредоточимся только на тех, которые устоялись как шаблоны.

Дополнительная информация

- Принципы проектирования контейнерных приложений (<https://red.ht/2HBKqYI>).
- Методология «Двенадцать факторов» (<https://12factor.net/ru/>).
- Сайт книги «Domain-Driven Design: Tackling Complexity in the Heart of Software»⁹ (http://dddcommunity.org/book/evans_2003).
- Руководство «Container Best Practices» (<http://bit.ly/2TUyNTe>).
- Руководство «Best Practices for Writing Dockerfiles» (<https://dockr.ly/2TFZBaL>).
- Презентация «Container Patterns» (<http://bit.ly/2TFjsH2>).
- Руководство «General Container Image Guidelines» (<https://red.ht/2u6Ahvo>).
- Описание понятия пода (<https://kubernetes.io/docs/user-guide/pods/>).

⁶ Эванс Эрик. Предметно-ориентированное проектирование (DDD).: Структуризация сложных программных систем. М.: Вильямс, 2010. — *Примеч. пер.*

⁷ Деинициализированные контейнеры еще не реализованы, но есть предложение (<http://bit.ly/2TegEM7>), описывающее, как включить эту возможность в будущие версии Kubernetes. События жизненного цикла мы обсудим в главе 5 «Управляемый жизненный цикл».

⁸ В переводе с английского pod — это стручок, кокон или группа. В контексте Kubernetes Pod — это группа контейнеров, запускаемых как одно целое. — *Примеч. пер.*

[9](#) Предметно-ориентированное проектирование (DDD): Структуризация сложных программных систем. М.: Вильямс, 2010. — *Примеч. пер.*

Часть I. Основные паттерны

В первой части книги описываются основные принципы, которым должны соответствовать контейнерные приложения, чтобы считаться хорошо соответствующими облачным окружениям. Следование этим принципам поможет гарантировать возможность автоматического управления вашими приложениями на облачных платформах, таких как Kubernetes.

В следующих главах описываются паттерны, которые являются основными строительными блоками распределенных контейнерных приложений на основе Kubernetes:

- Глава 2 «Предсказуемые требования» рассказывает, почему каждый контейнер должен объявлять свой профиль ресурсов и ограничиваться объявленными в нем требованиями.
- Глава 3 «Декларативное развертывание» демонстрирует разные стратегии развертывания приложений, которые могут выполняться декларативным способом.
- Глава 4 «Проверка работоспособности» описывает API, который должен быть реализован в каждом контейнере, чтобы помочь платформе наблюдать за приложением и поддерживать его в работоспособном состоянии.
- Глава 5 «Управляемый жизненный цикл» рассказывает, зачем контейнеру нужна возможность читать события, поступающие от платформы, и реагировать на них.
- Глава 6 «Автоматическое размещение» представляет паттерн для распределения контейнеров в кластере Kubernetes с несколькими узлами.

Глава 2. Предсказуемые требования

Основа успешного развертывания и сосуществования приложений в общем облачном окружении зависит от определения и объявления требований приложений к ресурсам и зависимостям времени выполнения. Паттерн *Предсказуемые требования* определяет, как должны объявляться требования приложений, будь то жесткие зависимости времени выполнения или требования к ресурсам. Объявление требований крайне важно для Kubernetes — это позволит фреймворку подобрать для вашего приложения правильное место в кластере.

Задача

Kubernetes может управлять приложениями, написанными на разных языках программирования, если эти приложения можно запускать в контейнере. Однако разные языки имеют разные требования к ресурсам. Как правило, программы, написанные на компилируемых языках, работают быстрее и часто требуют меньше памяти по сравнению с динамически компилируемыми программами или программами, выполняющимися под управлением интерпретатора. Учитывая, что многие современные языки программирования из одной и той же категории имеют схожие требования к ресурсам, с точки зрения потребления ресурсов более важными аспектами являются предметная область, бизнес-логика приложения и фактические детали реализации.

Трудно предсказать количество ресурсов, которое может понадобиться контейнеру для оптимального функционирования, и именно разработчик знает ожидаемый

объем ресурсов, необходимый для реализации службы (выявленный в ходе тестирования). Некоторые службы имеют постоянный профиль использования процессора и памяти, а некоторые — переменчивый. Некоторые службы нуждаются в долговременном хранилище для хранения данных; некоторые устаревшие службы требуют доступа к фиксированным портам в хост-системе для корректной работы. Описание всех этих характеристик приложений и передача их управляющей платформе – фундаментальное условие для нормального функционирования облачных приложений.

Помимо требований к ресурсам, среда времени выполнения приложений также зависит от возможностей платформы, таких как хранение данных или конфигурация приложения.

Решение

Знать требования контейнера к окружению времени выполнения важно по двум основным причинам. Во-первых, зная все зависимости времени выполнения и потребности в ресурсах, Kubernetes сможет принимать разумные решения о том, где в кластере разместить контейнер, чтобы максимально эффективно использовать оборудование. В окружении с общими ресурсами, используемыми большим числом процессов с разным приоритетом, знание требований каждого процесса является залогом их успешного сосуществования. Однако эффективное размещение — это только одна сторона медали.

Вторая причина важности профилей ресурсов контейнеров — это планирование вычислительных мощностей. Исходя из потребности в каждой конкретной службе и общего количества служб, можно оценить необходимые вычислительные мощности для разных окружений и определить экономически

наиболее эффективные профили хостов для удовлетворения потребностей всего кластера. Профили ресурсов служб и планирование вычислительной мощности вместе являются залогом долгого и успешного управления кластером.

Прежде чем углубиться в профили ресурсов, посмотрим, как объявлять зависимости времени выполнения.

Зависимости времени выполнения

Одной из наиболее типичных зависимостей времени выполнения является наличие хранилища файлов для сохранения состояния приложения. Контейнерные файловые системы эфемерны и исчезают после остановки контейнера. Для долговременного хранения файлов в подах (группах контейнеров) Kubernetes предлагает использовать тома.

Самый простой тип томов — `emptyDir`, который существует, пока существует использующий его под (Pod), то есть группа контейнеров, а после остановки пода его содержимое теряется. Том должен основываться на каком-то другом механизме хранения, чтобы данные в нем сохранялись после перезапуска пода. Если приложение нуждается в сохранении файлов в таком долговременном хранилище, нужно явно объявить эту зависимость в определении контейнера, как показано в листинге 2.1.

Листинг 2.1. Зависимость от PersistentVolume

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    - image: k8spatterns/random-generator:1.0
```

```
name: random-generator
volumeMounts:
- mountPath: "/logs"
  name: log-volume
volumes:
- name: log-volume
  persistentVolumeClaim: ❶
    claimName: random-generator-log
```

❶ Зависимость от наличия и привязки PVC

Планировщик проверит тип тома, который влияет на выбор места для размещения пода. Если поду нужен том, которого нет ни на одном узле кластера, тогда он вообще не будет планироваться для выполнения. Тома — это пример зависимости времени выполнения, которая влияет на выбор инфраструктуры для запуска пода и определения возможности запланировать его.

Аналогичная зависимость возникает, когда вы требуете от Kubernetes предоставить контейнеру определенный порт в хост-системе через `hostPort`. Объявление `hostPort` создает еще одну зависимость времени выполнения и ограничивает круг хостов, на которые может планироваться под. `hostPort` резервирует порт на каждом узле в кластере, из-за чего на каждом узле может быть запущено не более одного экземпляра пода. Из-за конфликтов портов возможности масштабирования таких подов ограничиваются количеством узлов в кластере Kubernetes.

Другой тип зависимости — конфигурации. Почти каждому приложению требуется некоторая информация о конфигурации, и для этой цели рекомендуется использовать карты конфигураций (ConfigMap), предлагаемые фреймворком Kubernetes. Ваши службы должны определить стратегию

использования конфигураций — либо через переменные среды, либо через файловую систему. В любом случае для вашего контейнера появляется зависимость времени выполнения от именованных карт конфигураций. Если будут созданы не все требуемые карты, контейнеры будут планироваться для выполнения на узле, но не будут запускаться. Карты конфигураций (ConfigMap) и секреты (Secret) более подробно описываются в главе 19 «Конфигурация в ресурсах», а в листинге 2.2 показано, как эти ресурсы используются в качестве зависимостей времени выполнения.

Листинг 2.2. Зависимость от ConfigMap

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: PATTERN
      valueFrom:
        configMapKeyRef: ❶
          name: random-generator-config
          key: pattern
```

❶ Зависимость от наличия ConfigMap.

Похожими на карты конфигураций являются секреты (Secret), предлагающие более безопасный способ передачи в контейнер специализированных конфигураций.[10](#) Секреты

используются точно так же, как карты конфигураций, и вводят такую же зависимость контейнера от пространства имен.

Создание объектов ConfigMap и Secret — это простые административные задачи, но хранилища и порты должны предоставляться узлами кластера. Некоторые из этих зависимостей сужают круг узлов, на которые может планироваться под (если вообще может), а другие могут препятствовать запуску пода. При разработке контейнерных приложений с такими зависимостями всегда учитывайте ограничения времени выполнения, которые они создадут позже.

Профили ресурсов

Объявить зависимость контейнера от карт конфигураций, секретов и томов несложно. Но чтобы выяснить требования контейнера к ресурсам, необходимо поразмышлять и поэкспериментировать. Вычислительные ресурсы в контексте Kubernetes — это все, что можно запросить, получить и использовать из контейнера. Ресурсы делятся на две группы: *сжимаемые* (могут регулироваться, например процессорное время или пропускная способность сети) и *несжимаемые* (нерегулируемые, например объем памяти).

Важно различать сжимаемые и несжимаемые ресурсы. Если контейнеры потребляют слишком много сжимаемых ресурсов, таких как процессор, их аппетиты урезаются, но если они используют слишком много несжимаемых ресурсов (например, памяти), они останавливаются (потому что нет другого способа попросить приложение освободить выделенную память).

Исходя из характера приложения и особенностей его реализации, нужно указать минимально необходимые объемы ресурсов (так называемые *запросы*) и максимально возможные

— до которых потребление может вырасти (*лимиты*). Определение контейнера может задавать необходимую долю процессорного времени и объем памяти в форме запроса и лимита. В общих чертах идея запросов/лимитов напоминает мягкие/жесткие лимиты. Например, аналогичным образом мы определяем размер кучи для приложения Java с помощью параметров командной строки `-Xms` и `-Xmx`.

При размещении подов на узлах планировщик ориентируется на величину запросов (а не лимитов). Планируя каждый конкретный под, планировщик рассматривает только узлы, имеющие достаточный объем ресурсов для размещения этого пода и всех его контейнеров, складывая запрашиваемые объемы ресурсов. В этом смысле поле запросов `requests` в определении контейнера влияет на возможность планирования пода. В листинге 2.3 показано, как такие ограничения определяются для подов.

Листинг 2.3. Ограничения ресурсов

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    resources:
      requests: ❶
        cpu: 100m
        memory: 100Mi
      limits: ❷
        cpu: 200m
```


memory: 200Mi

- ❶ Минимальные запросы процессорного времени и памяти.
- ❷ Верхние пределы, до которых может вырасти потребление приложения.

В зависимости от того, что вы определили — запросы, лимиты или и то и другое, — платформа предлагает разные уровни качества обслуживания (Quality of Service, QoS).

Без гарантий

Под, не определяющий запросы и лимиты для своих контейнеров, считается самым низкоприоритетным и уничтожается первым, когда на узле, где он располагается, заканчиваются несжимаемые ресурсы.

С переменным качеством

Под, определяющий неравные значения в запросах и лимитах (когда лимиты превышают запросы), гарантированно получает минимальный объем ресурсов, но при наличии свободных ресурсов может получить их больше, вплоть до объявленного предела. Когда узел испытывает нехватку ресурсов, эти поды могут останавливаться, если не останется подов, обслуживаемых без гарантий качества.

Гарантированный

Под, определяющий равные значения в запросах и лимитах, получает наивысший приоритет и гарантированно будут останавливаться, только если на узле не останется подов, обслуживаемых без гарантий или с переменным качеством.

Как видите, характеристики потребления ресурсов контейнерами, которые вы объявляете или опускаете, напрямую влияют на качество обслуживания и определяют относительную важность пода в случае истощения ресурса. Определяйте требования подов к ресурсам с учетом этого.

Приоритет пода

Мы выяснили, как объявление потребностей в ресурсах влияет на качество обслуживания и на очередность, в которой Kubelet¹¹ останавливает контейнеры в случае нехватки ресурсов. На момент написания этих строк уже велись работы по реализации еще одной возможности, имеющей отношение к качеству обслуживания и позволяющей определить приоритет пода и возможность его вытеснения. Настройка приоритета дает возможность указать важность пода относительно других подов и повлиять на порядок, в котором они будут планироваться. Давайте посмотрим на эти настройки, показанные в листинге 2.4.

Листинг 2.4. Приоритет пода

```
apiVersion: scheduling.k8s.io/v1beta1
kind: PriorityClass
metadata:
  name: high-priority      ❶
  value: 1000             ❷
  globalDefault: false
description: Это класс подов с очень высоким
приоритетом
---
apiVersion: v1
kind: Pod
metadata:
```

```
name: random-generator
labels:
  env: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    priorityClassName: high-priority ❸
```

- ❶ Имя объекта, определяющего класс приоритета.
- ❷ Значение приоритета в объекте.
- ❸ Класс приоритета, присвоенный этому поду, как определено в ресурсе PriorityClass.

Мы создали PriorityClass, объект вне пространства имен, чтобы определить целочисленный приоритет. Присвоили объекту PriorityClass имя high-priority и уровень приоритета 1000. Имея такое определение, мы можем назначить этот приоритет подам, указав имя объекта в параметре: `priorityClassName: high-priority`. PriorityClass — это механизм объявления важности подов по отношению друг к другу, где более высокое значение соответствует большей важности.

Когда возможность определения приоритетов будет полностью реализована, с ее помощью можно будет влиять на порядок размещения планировщиком подов на узлах. Контроллер учета приоритета сначала установит значения приоритетов в новых подах, используя поле `priorityClassName`. Когда появится несколько подов, ожидающих размещения, планировщик отсортирует очередь по убыванию приоритетов, благодаря чему более важные ожидающие поды будут выбираться из очереди планирования

раньше менее важных и, в отсутствие ограничений, препятствующих планированию, размещаться на узлах.

Здесь есть один важный момент. В отсутствие узлов с достаточным объемом ресурсов для размещения пода планировщик может выгрузить (удалить) поды с более низким приоритетом, чтобы освободить ресурсы и разместить модули с более высоким приоритетом. В результате под с более высоким приоритетом может быть запланирован раньше, чем под с более низким приоритетом, если выполнятся все другие требования планирования. Этот алгоритм позволяет администраторам кластера определять важность подов и гарантировать их первоочередное размещение, давая планировщику возможность удалять поды с более низким приоритетом, чтобы освободить место для подов с более высоким приоритетом. Если под не может быть запланирован, планировщик продолжит размещать другие поды, с более низким приоритетом.

Качество обслуживания (обсуждалось выше) и приоритет пода — это две независимые возможности. Управление качеством обслуживания используется главным образом агентами Kubelet для сохранения стабильности узла, когда доступные вычислительные ресурсы невелики. Выбирая, какой под остановить, Kubelet сначала оценивает его уровень качества обслуживания, а затем PriorityClass. С другой стороны, логика вытеснения в планировщике полностью игнорирует качество обслуживания при выборе подов для остановки. Планировщик пытается выбирать поды с наименьшим приоритетом, чтобы удовлетворить потребности подов с более высоким приоритетом, ожидающих размещения.

Объявление приоритета пода может оказать нежелательное влияние на другие поды, которые будут вытесняться. Например, несмотря на соблюдение политик правильного

завершения пода, выполнение требований бюджета нерработоспособности пода PodDisruptionBudget, о котором рассказывается в главе 10 «Служба-одиночка», не гарантируется, что может привести к нарушению работоспособности кластерного приложения с низким приоритетом, которое опирается на коллективную работу подов.

Другая проблема — пользователь, который по незнанию или по злему умыслу создает поды с наибольшим возможным приоритетом, мешающие выполнению всех остальных подов. Для предотвращения такой ситуации в механизм квотирования ресурсов ResourceQuota была добавлена поддержка PriorityClass и теперь большие значения приоритета зарезервированы для критических системных подов, которые обычно не должны вытесняться или останавливаться.

В заключение следует отметить, что приоритеты следует использовать с большой осторожностью, поскольку значения приоритетов, указанные пользователем и используемые планировщиком и агентами Kubelet, чтобы выяснить, какие поды разместить или остановить, часто выбираются экспериментально. Любые изменения могут повлиять на многие поды и помешать платформе соблюсти соглашение об уровне обслуживания.

Ресурсы проекта

Kubernetes — это платформа самообслуживания, которая позволяет разработчикам запускать приложения в predeterminedных изолированных окружениях. Однако для нормальной работы многопользовательской платформе необходимы также определенные границы и средства управления, чтобы отдельные пользователи не смогли захватить все ресурсы платформы. Одним из таких

инструментов является квотирование ресурсов ResourceQuota. С его помощью можно ограничить совокупное потребление ресурсов в пространстве имен. С помощью квот ResourceQuota администраторы кластера могут ограничить общую сумму используемых вычислительных ресурсов (процессор, память) и хранилища. С их помощью также можно ограничить общее количество объектов (карт конфигураций, секретов, подов и служб), создаваемых в пространстве имен.

Еще одним полезным инструментом в этой области является диапазон ограничений LimitRange, который позволяет установить границы использования ресурсов каждого типа. Помимо минимальных и максимальных значений и значений по умолчанию, этот механизм также позволяет контролировать отношение между запросами и лимитами, которое также известно как *уровень перерасхода*. В табл. 2.1 показан пример, как можно выбирать возможные значения для запросов и лимитов.

Таблица 2.1. Границы запросов и лимитов

Тип	Ресурс	Мин	Макс	Лимит по умолчанию	Запрос по умолчанию	Отношение лимит/запрос
Контейнер	Процессор	500 мс	2	500 мс	250 мс	4
Контейнер	Память	250 Мбайт	2 Гбайт	500 Мбайт	250 Мбайт	4

Диапазоны ограничений LimitRange удобно использовать для управления профилями ресурсов контейнеров, чтобы не допустить появления контейнеров, которым требуется больше ресурсов, чем может дать узел кластера. С его помощью также можно помешать пользователям создавать контейнеры, потребляющие большое количество ресурсов и делающие узлы недоступными для других контейнеров. Учитывая, что запросы (но не лимиты) являются основной характеристикой

контейнера, которую планировщик использует для размещения, отношение лимит/запрос `LimitRequestRatio` позволяет контролировать разницу между запросами и лимитами. Большой разрыв между запросами и лимитами увеличивает вероятность чрезмерной нагрузки на узел и может снизить производительность приложения, когда многим контейнерам одновременно потребуется больше ресурсов, чем запрашивалось первоначально.

Планирование вычислительных мощностей

Учитывая, что в разных окружениях контейнеры могут иметь разные профили ресурсов и разное количество экземпляров, очевидно, что планирование вычислительной мощности для многоцелевой среды — сложная задача. Например, для оптимального использования оборудования в непромышленном кластере можно использовать в основном контейнеры с уровнями качества обслуживания «без гарантий» и «с переменным качеством». В таком динамическом окружении одновременно может запускаться и останавливаться большое число контейнеров, и даже если какой-то контейнер будет остановлен платформой из-за нехватки ресурсов, это не будет иметь серьезных последствий. В промышленном кластере, где требуется высокая стабильность и предсказуемость, могут преобладать контейнеры с гарантированным уровнем качества обслуживания и в меньшем числе — с переменным качеством обслуживания. Принудительная остановка контейнера в таком окружении почти наверняка является признаком необходимости увеличить вычислительную мощность кластера.

В табл. 2.2 перечислено несколько служб с их требованиями к процессорному времени и памяти.

Таблица 2.2. Пример планирования вычислительной мощности

Под	Запрос процессора	Лимит процессора	Запрос памяти	Лимит памяти	Экземпляров
A	500 мс	500 мс	500 Мбайт	500 Мбайт	4
B	250 мс	500 мс	250 Мбайт	1000 Мбайт	2
C	500 мс	1000 мс	1000 Мбайт	2000 Мбайт	2
D	500 мс	500 мс	500 Мбайт	500 Мбайт	1
Всего	4000 мс	5500 мс	5000 Мбайт	8500 Мбайт	9

В реальной жизни такая платформа, как Kubernetes, обычно используется потому, что существует множество служб, часть из которых предполагается вывести из эксплуатации, а часть все еще находится на стадии проектирования и разработки. Но даже в таком постоянно меняющемся окружении можно рассчитать общий объем ресурсов, необходимых для всех служб.

Не забывайте также, что в разных окружениях может действовать разное число контейнеров, а еще нужно оставить место для автоматического масштабирования, заданий сборки, инфраструктурных контейнеров и многого другого. Опираясь на эту информацию и сведения о поставщике инфраструктуры, вы сможете выбрать наиболее экономичные вычислительные экземпляры, обладающие необходимыми ресурсами.

Пояснение

Контейнеры удобно использовать не только для изоляции процессов, но и как формат упаковки. С соответствующими профилями ресурсов они также помогают успешно планировать вычислительные мощности. Проведите несколько начальных экспериментов, чтобы выяснить потребности в ресурсах для каждого контейнера, и используйте эту

информацию как основу для будущего планирования и прогнозирования.

Однако, что более важно, профили ресурсов дают приложениям возможность сообщить платформе Kubernetes информацию, необходимую для планирования и управления решениями. Если приложение не определяет никаких запросов или лимитов, тогда Kubernetes будет интерпретировать его контейнеры как черные ящики, которые можно останавливать при заполнении кластера. Поэтому для каждого приложения важно продумать и определить требования к ресурсам.

Теперь, узнав, как определять размеры приложений, перейдем к главе 3 «Декларативное развертывание», где познакомимся с некоторыми стратегиями установки и обновления приложений в Kubernetes.

Дополнительная информация

- Пример предсказуемых требований (<http://bit.ly/2CrT8FJ>).
- Порядок использования карт конфигураций ConfigMap (<http://kubernetes.io/docs/user-guide/configmap/>).
- | | |
|-------|----------|
| Квоты | ресурсов |
|-------|----------|

(<http://kubernetes.io/docs/admin/resourcequota/>).
- Статья «Kubernetes Best Practices: Resource Requests and Limits» (<http://bit.ly/2ueNUc0>).
- Настройка пределов процессорного времени и памяти для пода (<http://kubernetes.io/docs/admin/limitrange/>).
- Дополнительные возможности управления потреблением ресурсами (<http://bit.ly/2TKEYKz>).

- О приоритетах подов и возможности их вытеснения (<http://bit.ly/2OdVcU6>).
- Об уровнях качества обслуживания в Kubernetes (<http://bit.ly/2HGimUq>).

[10](#) О защищенности секретов (Secret) мы подробно поговорим в главе 19 «Конфигурация в ресурсах».

[11](#) Агент кластера Kubernetes, который выполняется на каждом узле. — *Примеч. пер.*

Глава 3. Декларативное развертывание

Основой паттерна *Декларативное развертывание* является ресурс развертывания `Deployment`. Эта абстракция инкапсулирует процессы обновления и отката группы контейнеров и обеспечивает повторимость и автоматизацию развертывания.

Задача

Мы можем настроить изолированные окружения в виде пространств имен и размещать службы в этих окружениях с минимальным участием человека благодаря услугам планировщика. Но с ростом числа микросервисов постоянное обновление и замена их новыми версиями становятся все более трудоемкими.

Обновление службы до следующей версии включает такие действия, как запуск новой версии пода, безопасная остановка старой версии пода, ожидание и проверка успешности запуска, а иногда, в случае неудачи, откат до предыдущей версии. Эти действия выполняются либо ценой простоя, когда не работает ни одна из версий службы, либо без простоев, но за счет увеличения использования ресурсов двумя версиями службы, действующими одновременно в период обновления. Выполнение этих шагов вручную может привести к человеческим ошибкам и потребовать значительных усилий для правильного выполнения сценариев, что быстро превращает процесс выпуска новой версии в узкое место.

Императивное развертывание обновлений с помощью `kubectl` устарело

Фреймворк Kubernetes с самого начала поддерживал возможность обновления. Первая реализация была *императивной* по своей природе; клиент `kubectl` сообщал серверу, что делать на каждом шаге обновления.

Хотя команда `kubectl roll-update` все еще поддерживается, пользоваться ею не рекомендуется из-за следующих недостатков такого императивного подхода:

Вместо описания желаемого конечного состояния `kubectl rolling-update` выполняет команды, чтобы привести систему в нужное состояние.

Все управление заменой контейнеров и контроллеров репликации (ReplicationControllers) выполняется клиентом `kubectl`, который взаимодействует с API сервера, пока протекает процесс обновления, перекладывая на клиента всю ответственность, которую, по идее, должен нести сервер.

Может потребоваться более одной команды, чтобы привести систему в нужное состояние. Эти команды необходимо автоматизировать и повторять в разных окружениях.

Кто-то другой позже сможет затереть ваши изменения.

Процесс обновления требуется задокументировать и постоянно обновлять по мере развития службы.

Единственный способ узнать, что развертывание состоялось, — проверить состояние системы. Иногда состояние текущей

системы может не соответствовать желаемому, и тогда его следует отразить в документации с описанием процедуры развертывания.

Вместо процедуры следует использовать новый объект ресурса Deployment, который поддерживает *декларативное обновление*, полностью управляемое серверной стороной Kubernetes. Поскольку декларативные обновления имеют так много преимуществ, а поддержка императивного обновления со временем исчезнет, мы сосредоточимся исключительно на декларативных обновлениях в этом паттерне.

Решение

К счастью, Kubernetes предлагает средства автоматизации этой деятельности. Используя понятие *развертывания*, можно описать, как должно обновляться приложение, применяя разные стратегии и настраивая разные аспекты процесса обновления. Если за один цикл выпуска (который, в зависимости от команды и проекта, может длиться от нескольких минут до нескольких месяцев) развертывание каждого экземпляра микросервиса выполняется несколько раз, тогда эта автоматизация, предлагаемая Kubernetes, поможет сэкономить немало усилий.

В главе 2 мы увидели, что для эффективной работы планировщику необходимы: достаточный объем ресурсов на хост-системе, соответствующие политики размещения и контейнеры с правильно определенными профилями ресурсов. Аналогично, чтобы механизм развертывания успешно справился со своей работой, контейнеры должны быть сформированы для выполнения в облачном окружении. В

основе механизма развертывания, как нетрудно догадаться, лежит способность запускать и останавливать наборы подов. Чтобы этот механизм работал должным образом, сами контейнеры обычно должны принимать и обрабатывать события жизненного цикла (такие, как SIGTERM; см. главу 5 «Управляемый жизненный цикл»), а также предоставлять конечные точки для проверки их работоспособности, как описано в главе 4 «Проверка работоспособности».

Если контейнеры в точности выполняют эти два требования, платформа сможет чисто закрыть старые контейнеры и заменить их, запустив обновленные экземпляры. В таком случае все остальные аспекты процесса обновления можно определить декларативным способом и выполнить как одно атомарное действие с predetermined шагами и ожидаемым результатом. Давайте рассмотрим варианты обновления контейнеров.

Непрерывное развертывание

Декларативный способ обновления приложений в Kubernetes производится с использованием понятия развертывания. Внутренне механизм развертывания создает набор реплик ReplicaSet, который поддерживает селекторы меток. Кроме того, абстракция развертывания позволяет определить поведение процесса обновления с использованием таких стратегий, как RollingUpdate (непрерывное обновление, используется по умолчанию) и Recreate (воссоздание). В листинге 3.1 показаны важные детали настройки для стратегии непрерывного обновления.

Листинг 3.1. Непрерывное развертывание обновления

```
apiVersion: apps/v1
kind: Deployment
```

```

metadata:
  name: random-generator
spec:
  replicas: 3                                ❶
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1                            ❷
      maxUnavailable: 1                     ❸
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
        - image: k8spatterns/random-
generator:1.0
          name: random-generator
          readinessProbe:                    ❹
            exec:
              command: [ "stat", "/random-
generator-ready" ]

```

❶ Объявление трех реплик. Для непрерывного обновления необходимо больше одной реплики.

❷ Число подов, которые могут временно запускаться в дополнение к репликам, созданным для обновления. В этом примере максимальное число реплик равно четырем.

③ Число подов, которые могут быть недоступны в ходе обновления. В данном случае только два пода могут быть недоступны в ходе обновления.

④ Точки входа для проверки работоспособности (readiness probes) необходимы для развертывания без простоев — не забудьте о них (см. главу 4 «Проверка работоспособности»).

Стратегия `RollingUpdate` гарантирует отсутствие простоев в процессе обновления. Внутри реализация развертывания выполняет аналогичные действия, создавая новые наборы реплик и заменяя старые контейнеры новыми. Одним из плюсов использования механизма развертывания является возможность контролировать скорость развертывания нового контейнера. Объект `Deployment` позволяет управлять диапазоном доступных и избыточных подов через параметры `maxSurge` и `maxUnavailable`. На рис. 3.1 показано, как протекает процесс непрерывного обновления.

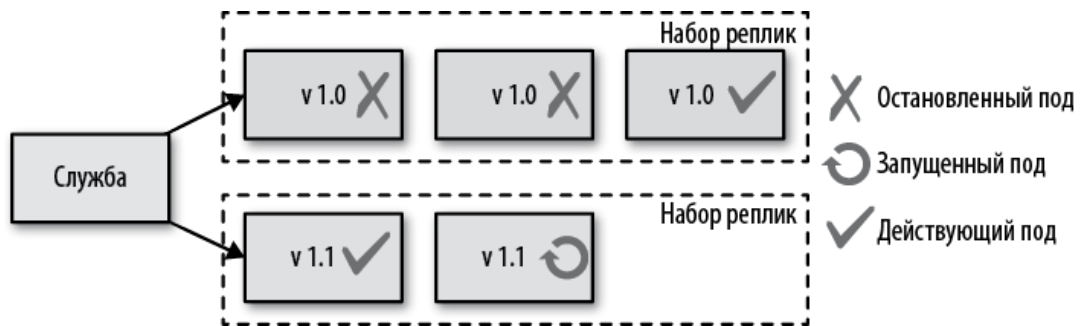


Рис. 3.1. Непрерывное развертывание

Запустить декларативное обновление можно тремя способами:

- Заменить все определение развертывания `Deployment` новой версией определения с помощью команды `kubectl replace`.

- Наложить исправления (`kubectl patch`) или внести правки интерактивно (`kubectl edit`), чтобы установить новый образ контейнера с новой версией.
- С помощью `kubectl set image` установить новый образ в развертывание `Deployment`.

Смотрите также полный пример (<http://bit.ly/2Fc6d6J>) в нашей репозитории, который демонстрирует использование этих команд и то, как можно отслеживать или откатывать обновления с помощью `kubectl rollout`.

Помимо устранения ранее упомянутых недостатков императивного способа развертывания служб, механизм развертывания `Deployment` дает следующие преимущества:

- `Deployment` — это объект ресурса Kubernetes, состояние которого полностью управляется фреймворком Kubernetes. Весь процесс обновления выполняется на стороне сервера, без взаимодействия с клиентом.
- Декларативный характер механизма развертывания `Deployment` позволяет увидеть конечное состояние после развертывания, а не шаги, необходимые для его достижения.
- `Deployment` — это выполняемый объект, опробованный и протестированный в нескольких окружениях, прежде чем он достиг состояния, готового к использованию в промышленном окружении.
- Процесс обновления также полностью записывается и имеет средства для приостановки, возобновления и отката к предыдущим версиям.

Фиксированное развертывание

Стратегия RollingUpdate гарантирует отсутствие простоя во время обновления. Но в ходе обновления одновременно действуют две версии контейнера, что является побочным эффектом. Это может вызвать проблемы у потребителей услуг, особенно когда в процессе обновления устанавливаются изменения в API, несовместимые со старой версией, а клиент не способен обработать эту ситуацию у себя. Для таких сценариев существует стратегия воссоздания Recreate, которая показана на рис. 3.2.

Стратегия Recreate заключается в присваивании параметру `maxUnavailable` числа объявленных реплик. То есть, согласно этой стратегии, сначала останавливаются все контейнеры с текущей версией, а затем запускаются сразу все новые контейнеры. В результате имеет место некоторый период

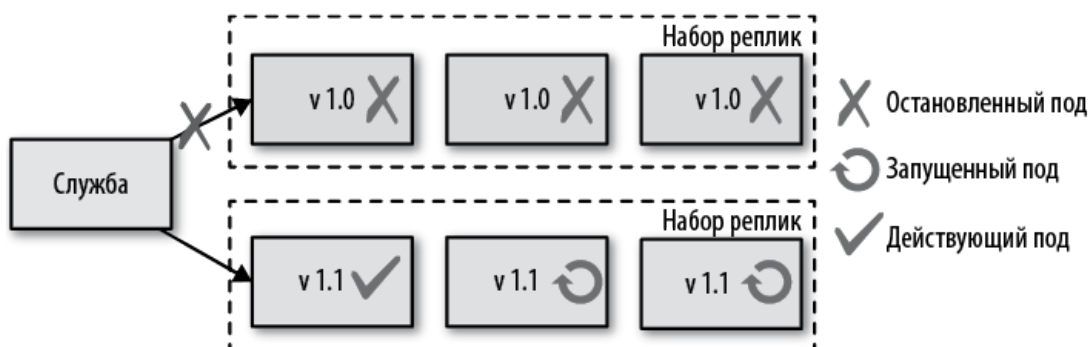


Рис. 3.2. Фиксированное развертывание с использованием стратегии Recreate

простоя, когда все контейнеры со старыми версиями уже остановлены, а новые еще не запущены или не готовы к приему и обработке запросов. Положительным моментом является тот факт, что одновременно не будут работать контейнеры двух версий, что упростит жизнь потребителям услуг.

Сине-зеленое развертывание

Сине-зеленое развертывание (Blue-Green deployment) — это стратегия, используемая для развертывания программного обеспечения в промышленном окружении, позволяющая минимизировать время простоя и уменьшить риски. Абстракция развертывания в Kubernetes — основополагающая идея, которая позволяет определить, как Kubernetes переходит от использования контейнеров одной версии к другой. Для реализации этой более продвинутой стратегии сине-зеленого развертывания можно использовать объект Deployment в комплексе с другими объектами Kubernetes.

Без использования таких расширений, как Service Mesh и Knative, сине-зеленое развертывание придется выполнять вручную. С технической точки зрения суть заключается в создании второго развертывания с последней версией контейнеров (назовем это *зеленым*), пока не обслуживающих никаких запросов. На этом этапе продолжают работать и обслуживать запросы старые реплики подов (называемые *синими*).

Убедившись, что новая версия подов работоспособна и готова обрабатывать запросы, весь трафик переключается на новые реплики. В Kubernetes это можно выполнить, изменив селектор службы, чтобы он соответствовал новым контейнерам (отмечены как *зеленые*). Как показано на рис. 3.3, когда весь трафик начинает поступать в зеленые контейнеры, синие контейнеры можно удалить, а ресурсы освободить для будущих сине-зеленых развертываний.

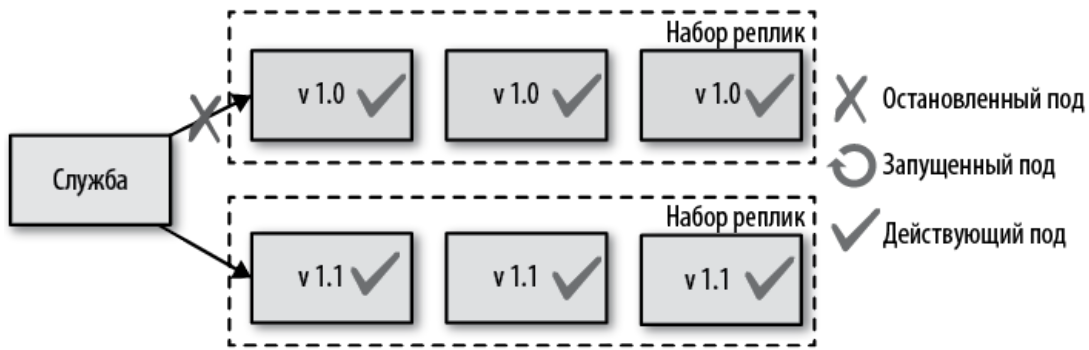


Рис. 3.3. Сине-зеленое развертывание

Преимущество сине-зеленого развертывания в том, что в каждый конкретный момент времени запросы обслуживает только одна версия приложения, что снижает сложность поддержки нескольких версий на стороне потребителей услуги. Но кластер должен иметь вдвое большую емкость, потому что в период развертывания в нем одновременно выполняются как синие, так и зеленые контейнеры. Кроме того, во время переходов могут возникнуть значительные осложнения с продолжительными процессами и изменениями в состоянии базы данных.

Канареечное развертывание

Канареечное развертывание (Canary deployment) — это способ постепенного развертывания новой версии приложения в рабочем окружении путем замены старых экземпляров новыми с помощью небольших блоков. Этот метод снижает риски, связанные с внедрением новой версии, открывая доступ к новым версиям только некоторым пользователям. Если новая версия хорошо зарекомендовала себя при обслуживании небольшой выборки пользователей, производится замена всех старых экземпляров. На рис. 3.4 показано, как выполняется канареечное развертывание.

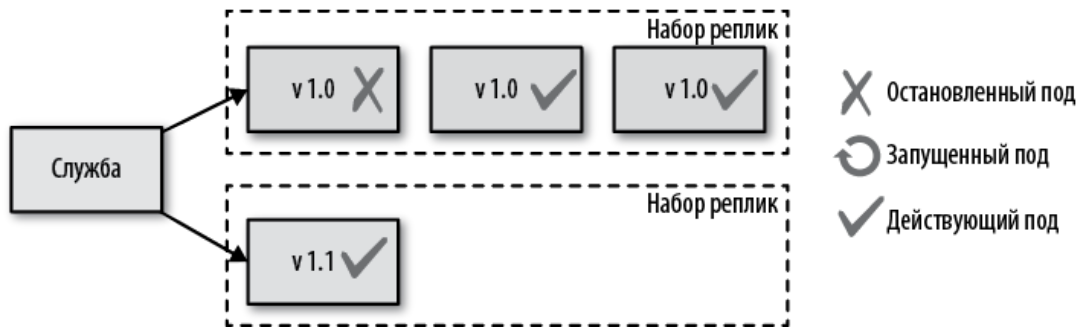


Рис. 3.4. Канареечное развертывание

В Kubernetes этот метод можно реализовать, создав набор реплик ReplicaSet для новой версии контейнера (предпочтительно с использованием объекта развертывания Deployment) с небольшим количеством реплик, который можно использовать в роли канареечного экземпляра. На этом этапе служба должна отсылать запросы некоторых пользователей в обновленные экземпляры подов. Убедившись, что новый набор реплик ReplicaSet действует должным образом, его можно масштабировать вверх, а старый набор — вниз, до нуля. В некотором смысле выполняется контролируемое и проверенное пользователем пошаговое развертывание.

Пояснение

Объект развертывания Deployment наглядно показывает, как Kubernetes превращает утомительный процесс ручного обновления приложений в декларативное действие, которое можно повторять и автоматизировать. Стратегии развертывания, поддерживаемые изначально (непрерывного обновления и воссоздания), управляют заменой старых контейнеров новыми, а стратегии выпуска (сине-зеленая и канареечная) определяют, как открывается доступ потребителям к новой версии. Последние две стратегии выпуска основаны на решении о переходе, принимаемом

человеком и, как следствие, автоматизированы не полностью и требуют взаимодействия с человеком. На рис. 3.5 показано, как меняется количество экземпляров новой и старой версий во время переходов при использовании разных стратегий развертывания и выпуска.

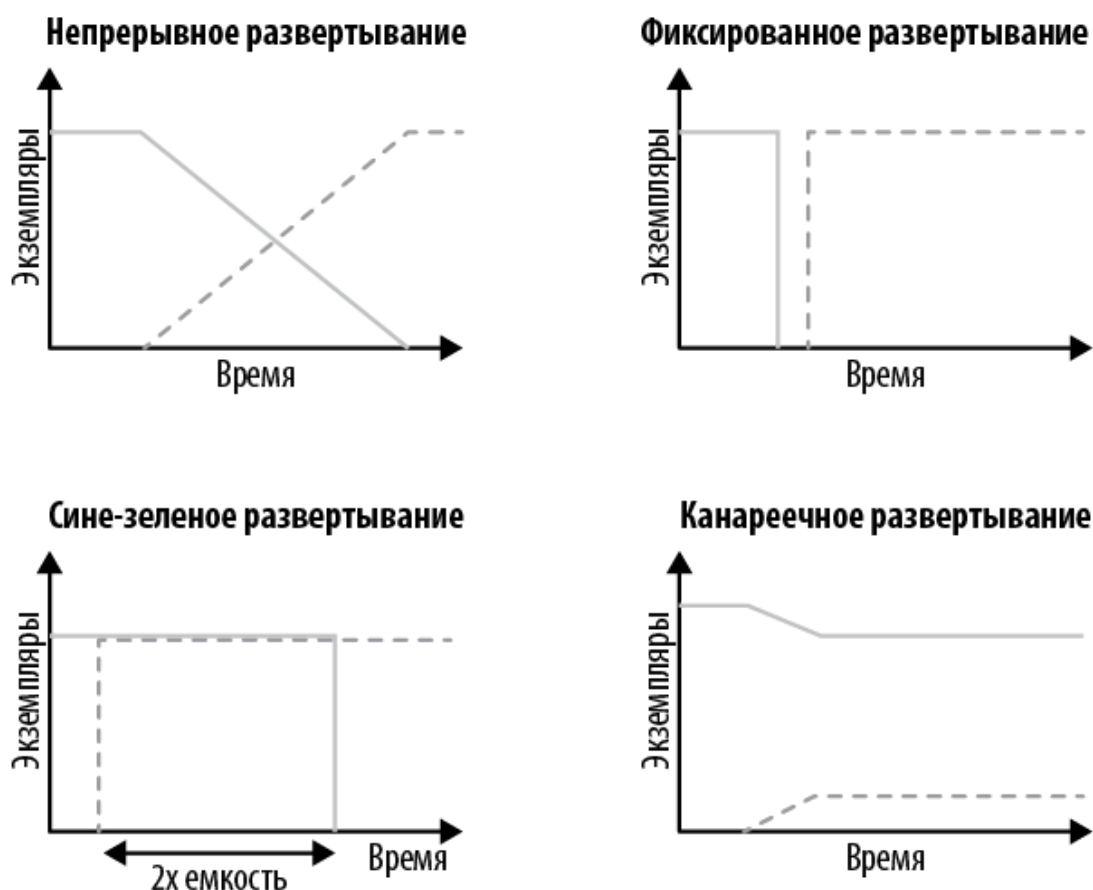


Рис. 3.5. Стратегии развертывания и выпуска

Каждый программный продукт уникален, и развертывание сложных систем часто требует дополнительных шагов и проверок. Методы, представленные в этой главе, охватывают процесс обновления подов, но не включают обновление и откат других зависимостей, таких как карты конфигураций ConfigMap, секретов Secret и других зависимых служб.

На момент написания этой книги было предложено реализовать в Kubernetes возможность вызова дополнительных

обработчиков в процессе развертывания. Обработчики Pre и Post, как предполагалось, должны позволить выполнять пользовательские команды до и после выполнения стратегии развертывания. Эти команды могли бы выполнять дополнительные действия во время развертывания и при необходимости прерывать, повторять или продолжать развертывание. Они могли бы стать первым шагом к созданию новых стратегий автоматического развертывания и выпуска. На данный момент используется подход, заключающийся в создании сценария процесса обновления на более высоком уровне, который управляет процессом обновления служб и их зависимостей с использованием объекта развертывания Deployment и других примитивов, обсуждаемых в этой книге.

Независимо от выбранной стратегии развертывания, чтобы выполнить требуемую последовательность шагов для достижения конечного состояния, Kubernetes должен знать, что поды приложения запустились и выполняются. Поэтому далее, в главе 4, мы рассмотрим паттерн *Health Probe* (Проверка работоспособности), используя который ваше приложение может сообщать Kubernetes о своем состоянии.

Дополнительная информация

- Пример декларативного развертывания (<http://bit.ly/2Fc6d6J>).
- Непрерывное обновление (<http://bit.ly/2r06lch>).
- Описание объекта Deployment (<http://bit.ly/2q7vR7Y>).
- Запуск приложения без состояния с помощью Deployment (<http://bit.ly/2XZZhLL>).

- Статья Мартина Фаулера (Martin Fowler) «Blue-Green Deployment»¹² (<http://bit.ly/1Gph4FZ>).
- Статья Данило Сато (Danilo Sato) «Canary Release» (<https://martinfowler.com/bliki/CanaryRelease.html>).
- Книга «DevOps with OpenShift» (<https://red.ht/2W7fdAQ>).

¹² Перевод на русский язык доступен по адресу <https://habr.com/ru/post/309832/>.
— *Примеч. пер.*

Глава 4. Проверка работоспособности

Паттерн *Health Probe* (Проверка работоспособности) определяет порядок передачи информации о состоянии приложения в Kubernetes. Чтобы добиться максимальной автоматизации, облачное приложение должно постоянно сообщать о своем состоянии, чтобы фреймворк Kubernetes мог определить, работает ли приложение и готово ли оно обслуживать запросы. Доступность этой информации влияет на управление жизненным циклом подов и способом передачи трафика в приложение.

Задача

Kubernetes регулярно проверяет состояние процесса контейнера и перезапускает его при обнаружении проблемы. Однако из практики мы знаем, что простой проверки состояния процесса недостаточно для определения работоспособности приложения. Часто бывает так, что приложение зависает, а его процесс все еще работает. Например, приложение на Java может столкнуться с ошибкой *OutOfMemoryError* (нехватка памяти), но при этом процесс JVM будет продолжать выполняться. Также приложение может зависнуть, попав в бесконечный цикл или в состояние взаимоблокировки или когда система начинает активно использовать файл подкачки из-за нехватки памяти. Для выявления подобных ситуаций фреймворку Kubernetes нужен надежный способ проверки работоспособности приложений — не для того, чтобы понять внутреннюю работу приложения, а чтобы убедиться, что приложение работает как ожидается и способно обслуживать потребителей.

Решение

Индустрия программного обеспечения признает невозможность писать код без ошибок. Более того, в распределенных приложениях вероятность сбоев возрастает. В результате основное внимание сместилось с обнаружения и устранения неисправностей на обнаружение сбоев и восстановление работоспособности после них. Обнаружение сбоя — сложная задача. Она не имеет универсального решения, пригодного для всех приложений, потому что разные приложения имеют разное понятие сбоя. Кроме того, разные виды сбоев требуют разных корректирующих воздействий. Временные сбои могут устраняться самим приложением при наличии достаточного времени, а при некоторых других сбоях может потребоваться перезапустить приложение. Давайте посмотрим, какие проверки использует Kubernetes для обнаружения и исправления сбоев.

Проверка наличия процесса

Проверка наличия процесса — это самая простая проверка, которую Kubelet выполняет постоянно. Если процессы не запущены, они запускаются. То есть благодаря только одной этой простой проверке приложение становится немного более надежным. Если ваше приложение способно самостоятельно обнаружить сбой и завершиться, тогда вам вполне достаточно будет этой простой проверки работоспособности процесса. Однако для большинства случаев этого недостаточно и необходимы другие виды проверок.

Проверка работоспособности

Если ваше приложение попадает в ситуацию взаимоблокировки, процесс не останавливается сам по себе и

продолжает действовать. Чтобы обнаружить подобные проблемы и любые другие нарушения в работе бизнес-логики приложения, Kubernetes использует *проверки работоспособности* — регулярные проверки, выполняемые агентом Kubelet, который запрашивает у контейнера подтверждение его работоспособности. Важно, чтобы проверка работоспособности выполнялась извне, а не внутри самого приложения, потому что некоторые сбои могут помешать сторожевому коду в приложении сообщить об ошибке. В отношении корректирующих воздействий проверка работоспособности аналогична проверке наличия процесса, поэтому при обнаружении сбоя контейнер перезапускается. Однако эти проверки обеспечивают большую гибкость в выборе методов проверки приложения, а именно:

- Проверка через HTTP заключается в отправке HTTP-запроса GET на IP-адрес контейнера и в ожидании HTTP-ответа с кодом от 200 до 399.
- Проверка через сокет TCP предполагает благополучный обмен через соединение TCP.
- Проверка через выполнение заключается в выполнении произвольной команды в ядре контейнера и в ожидании кода благополучного ее завершения (0).

Организация проверки работоспособности через HTTP показана в листинге 4.1.

Листинг 4.1. Контейнер с поддержкой проверки работоспособности

```
apiVersion: v1
kind: Pod
metadata:
```

```
name: pod-with-liveness-check
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: DELAY_STARTUP
      value: "20"
    ports:
    - containerPort: 8080
    protocol: TCP
    livenessProbe:
      httpGet:
        path: /actuator/health
        port: 8080
      initialDelaySeconds: 30
```

❶ Конечная точка для проверки через HTTP.

❷ Ждать 30 секунд перед первой проверкой работоспособности, чтобы дать приложению время запуститься.

В зависимости от характера приложения, вы можете подобрать наиболее подходящий метод. Решение о признании или непризнании приложения работоспособным зависит от конкретной реализации. Но имейте в виду, что если приложение будет признано неработоспособным, произойдет перезапуск контейнера. Если перезапуск контейнера не помогает, продолжать проверку работоспособности бессмысленно, потому что перезапуск не устраняет основную проблему.

Проверка готовности

Проверку работоспособности удобно использовать для поддержания приложений в работоспособном состоянии, останавливая сбойные контейнеры и заменяя их новыми. Но иногда контейнер может действовать с ошибками и его повторный запуск не решает проблему. Наиболее распространенный пример — когда контейнер запускается, но по каким-то причинам не готов обрабатывать какие-либо запросы. Или, может быть, контейнер перегружен, задержка в обработке запросов увеличивается, и вы хотите, чтобы он временно предпринял меры защиты против высокой нагрузки.

Для таких сценариев Kubernetes поддерживает *проверки готовности*. Для проверки готовности используют те же методы, что и для проверки работоспособности (через HTTP, TCP или выполнение команд), но корректирующие воздействия отличаются. Если проверка готовности потерпела неудачу, контейнер не перезапускается, а просто исключается из обработки входящего трафика. Проверка готовности помогает определить, когда контейнер будет готов к работе, чтобы дать ему некоторое время для инициализации перед получением запросов от службы. Также эти проверки могут использоваться для отключения службы от трафика на более поздних этапах, потому что проверки готовности выполняются регулярно, аналогично проверкам работоспособности. В листинге 4.2 показано, как организовать проверку готовности путем проверки наличия файла, который приложение создает по завершении подготовки к работе.

Листинг 4.2. Контейнер с поддержкой проверки готовности

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-readiness-check
```

```
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    readinessProbe:
      exec:
        ❶
        command: [ "stat", "/var/run/random-
generator-ready" ]
```

❶ Проверка наличия файла, создаваемого приложением по завершении подготовки к обработке запросов. `stat` возвращает ошибку, если файл отсутствует, что обеспечивает отрицательный результат проверки.

И снова ваша реализация проверки работоспособности должна решать, когда приложение готово к выполнению своей работы, а когда его следует оставить в покое. В отличие от проверки наличия процесса и проверки работоспособности, которые предназначены для восстановления после сбоя путем перезапуска контейнера, проверка готовности требует дать приложению время на восстановление. Помните, что Kubernetes попытается оградить ваш контейнер от получения новых запросов (например, когда он выключается) независимо от того, проходит ли проверка готовности после получения сигнала SIGTERM.

Во многих случаях у вас будут одинаковые проверки работоспособности и готовности. Но проверка готовности дает контейнеру время для запуска. Только после благополучного прохождения проверки готовности развертывание считается успешным, и, например, поды с более старой версией могут быть остановлены в процессе непрерывного обновления.

Проверки работоспособности и готовности являются основополагающими строительными блоками для

автоматизации облачных приложений. Прикладные фреймворки, такие как Spring, Wild-Fly Swarm, Karaf или MicroProfile для Java, предлагают свои реализации паттерна *Health Probes* (Проверка работоспособности).

Пояснение

Для полной автоматизации облачные приложения должны быть максимально открыты и предоставлять управляющей платформе средства чтения и интерпретации информации о работоспособности приложения и, при необходимости, для выполнения корректирующих воздействий. Проверка работоспособности играет важную роль в автоматизации таких действий, как развертывание, автоматическое восстановление, масштабирование и др. Однако существуют и другие средства, с помощью которых приложение может сообщить о себе больше информации.

Очевидный и старый метод решения этой задачи — журналирование. Желательно, чтобы контейнеры фиксировали в журналах все значимые события, связанные с ошибками, и хранили эти журналы в централизованном месте для дальнейшего анализа. Журналы, как правило, не используются для автоматизации действий, их задача — оповещение и помощь в дальнейшем расследовании. Более полезным аспектом журналов является возможность анализа причин сбоев и обнаружение незаметных ошибок.

Помимо вывода информации в стандартные потоки, также рекомендуется записывать причины завершения контейнера в */dev/termination-log*. Это место, куда контейнер может записать свои последние слова, прежде чем окончательно исчезнуть. На рис. 4.1 показаны возможные варианты взаимодействия контейнера с платформой времени выполнения.



Рис. 4.1. Варианты наблюдения за контейнером

Контейнеры предоставляют универсальный способ упаковки и запуска приложений, интерпретируя их как черные ящики. Однако любой контейнер, предназначенный для выполнения в облачном окружении, должен предоставить API, с помощью которого среда выполнения сможет наблюдать за состоянием контейнера и предпринимать соответствующие действия. Это является главной предпосылкой для автоматизации обновления контейнеров и управления их жизненным циклом обобщенным способом, что, в свою очередь, повышает устойчивость системы и удобство работы пользователей. С практической точки зрения контейнерное приложение как минимум должно предоставлять API для различных видов проверок (работоспособности и готовности).

Еще более эффективные приложения должны также предоставлять управляющей платформе дополнительные средства для наблюдения за своим состоянием посредством интеграции с библиотеками трассировки и сбора метрик, такими как OpenTracing или Prometheus. Относитесь к своему приложению как к черному ящику, но реализуйте все необходимые API, чтобы упростить наблюдение и управление вашим приложением.

Следующий паттерн, *Managed Lifecycle* (Управляемый жизненный цикл), также основывается на обмене данными между приложениями и управляющим слоем в Kubernetes, но

решает другую задачу — задачу получения приложением информации о важных событиях жизненного цикла пода.

Дополнительная информация

- Пример реализации проверки работоспособности (<http://bit.ly/2Y6wCLG>).
- Настройка проверок работоспособности и готовности (<http://bit.ly/2r096A3>).
- Настройка проверки наличия процесса с использованием проверок работоспособности и готовности (<http://bit.ly/2HJkoDf>).
- Ресурс качества обслуживания (Quality of Service) в Kubernetes (<http://bit.ly/2HGimUq>).
- Организация нормального завершения с Node.js и Kubernetes (<http://bit.ly/2udUfo0>).
- Дополнительные паттерны проверки работоспособности в Kubernetes (<https://ahmet.im/blog/advanced-kubernetes-health-checks/>).

Глава 5. Управляемый жизненный цикл

Контейнерные приложения, действующие в облачном окружении, не управляют своим жизненным циклом и поэтому должны слушать события, генерируемые управляющей платформой, и, соответственно, корректировать свои жизненные циклы. Паттерн *Managed Lifecycle* (Управляемый жизненный цикл) определяет, как приложения могут и должны реагировать на события жизненного цикла.

Задача

В главе 4 «Проверка работоспособности» мы узнали, почему контейнеры должны предоставлять программный интерфейс для различных проверок работоспособности. Программный интерфейс проверки работоспособности — это комплекс конечных точек, доступных только для чтения, которые платформа постоянно опрашивает, чтобы получить представление о приложении. Это — механизм, используемый платформой для извлечения информации из приложения.

Кроме мониторинга состояния контейнера, платформа иногда может выдавать команды, ожидая определенной реакции от приложения. Опираясь на внутренние правила и внешние факторы, облачная платформа может в любой момент принять решение о запуске или остановке управляемых ею приложений. Но только само контейнерное приложение может определить, на какие события и как оно будет реагировать. Фактически приложение предлагает программный интерфейс, который платформа использует для связи и отправки команд приложению. Кроме того, приложения могут получать дополнительные выгоды от

управления жизненным циклом извне или игнорировать управляющие воздействия, если эта услуга им не нужна.

Решение

Выше мы видели, что простая проверка статуса процесса — недостаточно хороший показатель работоспособности приложения. Вот почему существуют разные API для мониторинга работоспособности контейнера. Аналогично, использования одной только модели процесса для запуска и остановки приложения недостаточно. Часто приложения требуют более тонких воздействий и механизмов управления жизненным циклом. Некоторым приложениям нужна помощь для разогрева, а некоторым требуется выполнить точную и четкую процедуру завершения. Для этих и других случаев платформа генерирует некоторые события, как показано на рис. 5.1, которые контейнер может принимать и обрабатывать, если это необходимо.



Рис. 5.1. Управляемый жизненный цикл контейнера

Единицей развертывания приложения является под. Как вы уже знаете, под состоит из одного или нескольких контейнеров. На уровне пода имеются другие конструкции, такие как `init-контейнеры` (рассматриваются в главе 14 «`Init-контейнер`») и `defer-контейнеры` (контейнеры с отложенным запуском, находившиеся на стадии предложения на момент написания

этих строк), которые могут помочь в управлении жизненным циклом контейнера. Все события и точки входа, которые описываются в этой главе, применяются на уровне отдельного контейнера, а не на уровне пода.

Сигнал SIGTERM

Всякий раз, когда фреймворк Kubernetes решает остановить контейнер, например, останавливая под, которому тот принадлежит, или перед повторным запуском с целью устранения неисправности, выявленной при проверке работоспособности, контейнер получает сигнал SIGTERM. SIGTERM — это вежливое предложение контейнеру завершиться самому, прежде чем Kubernetes отправит более резкий «окрик» — сигнал SIGKILL. Получив сигнал SIGTERM, приложение должно завершиться как можно быстрее. Одни приложения могут быстро остановиться в ответ на этот сигнал, другим приложениям может потребоваться завершить уже запущенные запросы, закрыть соединения и очистить временные файлы, что может занять немного больше времени. В любом случае реакция на SIGTERM обозначает подходящий момент для аккуратного завершения контейнера.

Сигнал SIGKILL

Если процесс контейнера не остановился после сигнала SIGTERM, он принудительно завершается следующим сигналом SIGKILL. Kubernetes не посылает сигнал SIGKILL немедленно, а ожидает 30 секунд по умолчанию после отправки сигнала SIGTERM. Этот период можно настроить отдельно для каждого пода, `определив` параметр `.spec.terminationGracePeriodSeconds`, но соблюдение этой настройки не гарантируется, потому что ее можно

переопределить в командах Kubernetes. Поэтому разработчики должны стремиться проектировать и реализовать контейнерные приложения так, чтобы они быстро запускались и завершались.

Точка входа postStart

Однако одних только сигналов для управления жизненным циклом процесса недостаточно. Вот почему в Kubernetes существуют дополнительные обработчики событий жизненного цикла, такие как postStart и preStop. В листинге 5.1 показано определение пода с точкой входа postStart.

Листинг 5.1. Контейнер с обработчиком postStart

```
apiVersion: v1
kind: Pod
metadata:
  name: post-start-hook
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    lifecycle:
      postStart:
        exec:
          command:
            ❶
            - sh
            - -c
              - sleep 30 && echo "Wake up!" >
                /tmp/postStart_done
```

❶ Команда `postStart` в этом примере ждет 30 секунд. `sleep` здесь просто имитирует продолжительный процесс инициализации приложения. Кроме того, она использует файл для синхронизации с основным приложением, которое запускается параллельно.

Команда `postStart` выполняется после создания контейнера, параллельно с процессом самого контейнера. Даже при том, что логику инициализации и разогрева приложения часто можно реализовать часть процедуры запуска контейнера, обработчик `postStart` все еще может пригодиться в некоторых ситуациях. По своему характеру `postStart` является блокирующей операцией, и контейнер остается в состоянии *Waiting* (пауза) до завершения обработчика `postStart`, что, в свою очередь, заставляет всю группу контейнеров (под) оставаться в состоянии *Pending* (ожидание). Эту особенность `postStart` можно использовать, чтобы отложить инициализацию контейнера и дать время для инициализации основного процесса контейнера.

Другое применение `postStart` — предотвращение запуска контейнера при несоблюдении некоторых предварительных условий. Например, если обработчик `postStart` сообщит об ошибке, вернув ненулевой код завершения, фреймворк Kubernetes уничтожит основной процесс контейнера.

Механизмы вызова точек входа `postStart` и `preStop` напоминают вызов точек входа определения работоспособности, как описывалось в главе 4 «Проверка работоспособности», и поддерживают следующие типы обработчиков:

exec

Выполняет команду непосредственно в контейнере.

HttpGet

Выполняет HTTP-запрос GET, посылая его в порт, открытый одним из контейнеров в поде.

Будьте осторожны, закладывая критически важную логику в обработчик `postStart`, потому что нет никаких гарантий относительно порядка его выполнения. Поскольку обработчик выполняется параллельно с процессом контейнера, есть вероятность, что он выполнится до запуска контейнера. Кроме того, обработчик должен соответствовать семантике выполнения «не менее одного раза», то есть он должен позаботиться о попытках повторного запуска. Еще один аспект, о котором следует помнить, — платформа не выполняет повторных попыток, если HTTP-запрос не достиг обработчика.

Точка входа `preStop`

Обработчик `preStop` — это блокирующий запрос, посылаемый контейнеру перед его завершением. Он имеет ту же семантику, что и сигнал `SIGTERM`, и должен использоваться для запуска процедуры останова контейнера, когда реализовать перехват сигнала `SIGTERM` невозможно. Обработчик `preStop`, как показано в листинге 5.2, должен завершиться до того, как в среду выполнения контейнера будет послан запрос на удаление контейнера, который инициирует отправку сигнала `SIGTERM`.

Листинг 5.2. Контейнер с обработчиком `preStop`

```
apiVersion: v1
kind: Pod
metadata:
  name: pre-stop-hook
spec:
  containers:
```

```
- image: k8spatterns/random-generator:1.0
  name: random-generator
  lifecycle:
    preStop:
      httpGet: ❶
        port: 8080
        path: /shutdown
```

❶ Посылает запрос в конечную точку `/shutdown` приложения.

Но даже при том, что `preStop` имеет блокирующий характер, он не может предотвратить завершение контейнера, вернув признак ошибки, или удерживать его от остановки до бесконечности. Обработчик `preStop` — это лишь более удобная альтернатива сигналу `SIGTERM`, позволяющая организовать правильное завершение приложения, и ничего более. Он поддерживает те же типы обработчиков и гарантии, что и описанный выше обработчик `postStart`.

Другие средства управления жизненным циклом

В этой главе мы до сих пор рассматривали обработчики, позволяющие выполнять команды в ответ на события жизненного цикла контейнера. Но существует и другой механизм, находящийся не на уровне контейнера, а на уровне пода, который позволяет выполнять инструкции инициализации.

Мы подробно поговорим об этом механизме в главе 14 «Init-контейнер», а здесь лишь кратко опишем его, чтобы сравнить с обработчиками жизненного цикла. В отличие от обычных контейнеров приложений, `init`-контейнеры запускаются последовательно, выполняются до завершения и

запускаются перед любыми контейнерами приложений в поде. Эти гарантии позволяют использовать `init`-контейнеры для задач инициализации на уровне пода. Обработчики событий жизненного цикла и `init`-контейнеры работают на разных уровнях детализации (на уровне контейнера и на уровне пода соответственно) и могут использоваться взаимозаменяемо или дополнять друг друга. В табл. 5.1 перечислены основные отличия между ними.

Таблица 5.1. Обработчики событий жизненного цикла и `init`-контейнеры

Аспект	Обработчики событий жизненного цикла	<code>init</code> -контейнеры
Активируется в	Этапы жизненного цикла контейнера	Этапы жизненного цикла пода
На этапе запуска	Выполняется обработчик <code>postStart</code>	Выполняются контейнеры из списка <code>initContainers</code>
На этапе остановки	Выполняется обработчик <code>prestop</code>	Пока нет эквивалентного механизма
Гарантии относительно порядка выполнения	Обработчик <code>postStart</code> выполняется одновременно с <code>ENTRYPOINT</code> контейнера	Все <code>init</code> -контейнеры должны завершиться с признаком успеха до того, как будет запущен первый прикладной контейнер
Случаи использования	Для выполнения некритичных операций запуска/остановки, характерных для контейнера	Выполнение последовательности операций с использованием контейнеров; повторное использование контейнеров для выполнения задач

Нет никаких строгих правил, предписывающих, какой механизм использовать. Единственный критерий — необходимость конкретных гарантий относительно порядка выполнения. Можно полностью отказаться от использования обработчиков событий жизненного цикла и `init`-контейнеров и использовать сценарии `bash` для выполнения определенных действий на этапах запуска и завершения контейнеров. Это

возможно, но такой подход тесно связывает контейнер со сценарием и усложняет его сопровождение.

Можно ограничиться использованием обработчиков событий жизненного цикла Kubernetes для выполнения некоторых действий, как описано в этой главе, а можно пойти дальше и запускать контейнеры, которые выполняют отдельные действия с помощью `init`-контейнеров. Для организации такой последовательности требуется больше усилий, зато она предлагает более надежные гарантии и допускает повторное использование.

Знание этапов и доступных обработчиков событий жизненного цикла контейнеров и подов имеет решающее значение для создания приложений, которые получают дополнительные выгоды от выполнения под управлением Kubernetes.

Пояснение

Одним из основных преимуществ облачной платформы является возможность надежного и предсказуемого выполнения и масштабирования приложений в потенциально ненадежной облачной инфраструктуре. Эти платформы предлагают набор ограничений и контрактов для приложений, работающих под их управлением. В интересах приложения следовать этим контрактам, чтобы воспользоваться всеми возможностями, предлагаемыми облачной платформой. Обработка этих событий гарантирует правильный запуск и завершение приложения с минимальным воздействием на потребляющие их службы. На данный момент это означает, что контейнеры должны действовать подобно хорошо спроектированному процессу POSIX. В будущем могут появиться другие события, подсказывающие приложению,

когда оно будет масштабироваться, или предлагающие освободить ресурсы, чтобы предотвратить преждевременное завершение. Важно понимать, что жизненный цикл приложения больше не контролируется человеком, а полностью автоматизируется платформой.

Помимо управления жизненным циклом приложения, другой большой обязанностью платформ управления, таких как Kubernetes, является распределение контейнеров по массиву узлов. Паттерн *Automated Placement* (Автоматическое размещение) определяет приемы влияния на решения по планированию извне.

Дополнительная информация

- Пример организации управления жизненным циклом (<http://bit.ly/2udxws4>).
- Обработчики событий жизненного цикла контейнера (<http://bit.ly/2Fb38Uk>).
- Подключение обработчиков к событиям жизненного цикла контейнера (<http://bit.ly/2Jn9ANi>).
- Аккуратное завершение (<http://bit.ly/2TcPnJW>).
- Аккуратное завершение подов в Kubernetes (<http://bit.ly/2CvDQjs>).
- Defer-контейнеры (<http://bit.ly/2TegEM7>).

Глава 6. Автоматическое размещение

Автоматическое размещение является основной функцией планировщика Kubernetes, который распределяет новые поды между узлами в соответствии с требованиями контейнеров к ресурсам и с соблюдением правил планирования. Этот паттерн описывает принципы алгоритма планирования в Kubernetes и способы влияния на решения о размещении.

Задача

Типичная система на основе микросервисов состоит из десятков или даже сотен изолированных процессов. Контейнеры и поды служат хорошими абстракциями для упаковки и развертывания, но не решают проблему размещения этих процессов на подходящих узлах. При большом и постоянно растущем количестве микросервисов назначение и размещение их по отдельности начинает вызывать неуправляемое нарастание сложностей.

Контейнеры имеют зависимости друг от друга, зависимости от узлов и потребности в ресурсах, и все эти параметры меняются со временем. Ресурсы, доступные в кластере, также меняются со временем из-за сокращения или расширения кластера или из-за того, что часть ресурсов уже занята размещенными контейнерами. Порядок размещения контейнеров также влияет на доступность, производительность и емкость распределенных систем. Все это делает планирование контейнеров движущейся целью, в которую приходится стрелять на ходу.

Решение

В Kubernetes размещение подов на узлах осуществляется планировщиком. Это область, имеющая массу настроек, на момент написания этих строк все еще продолжала быстро развиваться и изменяться. В этой главе мы рассмотрим основные механизмы управления планированием, движущие силы, влияющие на размещение, а также причины и последствия выбора того или иного варианта. Планировщик Kubernetes является мощным инструментом, позволяющим экономить время. Он играет фундаментальную роль в платформе Kubernetes, но, как и другие компоненты Kubernetes (API Server, Kubelet), его можно использовать изолированно или вообще не использовать.

На самом верхнем уровне планировщик Kubernetes извлекает определение каждого вновь созданного пода, используя API Server, и связывает его с определенным узлом. Он отыскивает подходящий узел (если таковой имеется) для каждого пода, будь то первоначальное размещение приложения, масштабирование вверх или перемещение с вышедшего из строя узла на работоспособный узел. При этом учитываются зависимости времени выполнения, требования к ресурсам и высокой доступности, используются приемы горизонтального распределения подов и размещения подов поблизости друг от друга для уменьшения задержек при взаимодействиях. Однако чтобы планировщик правильно выполнял свою работу и допускал возможность декларативного размещения, ему нужны информация о емкости узлов и контейнеры с объявленными профилями ресурсов и действующими политиками. Давайте рассмотрим каждое из требований подробнее.

Доступные ресурсы на узле

Прежде всего, кластер Kubernetes должен иметь узлы с объемом ресурсов, достаточным для запуска новых подов. Каждый узел имеет определенную емкость для запуска подов, и планировщик гарантирует, что сумма ресурсов, запрашиваемых подом, не превысит доступную емкость узла. Емкость узла, выделенного только для нужд Kubernetes, рассчитывается по формуле в листинге 6.1.

Листинг 6.1. Емкость узла

$$\begin{aligned} \text{Доступная емкость [для подов приложений]} = & \\ & \text{Полная емкость узла [весь объем ресурсов узла]} \\ & - \text{Емкость для нужд Kubernetes [демонов Kubernetes, таких как} \\ & \text{kubelet, среды} \\ & \text{выполнения контейнера]} \\ & - \text{Емкость для нужд системы [демонов ОС,} \\ & \text{таких как sshd, udev]} \end{aligned}$$

Если не зарезервировать ресурсы для системных демонов, которые обеспечивают работу ОС и самого фреймворка Kubernetes, и отдать под нужды подов полную емкость узла, это может привести к тому, что поды и системные демоны будут конкурировать за ресурсы, что приведет к проблемам, связанным с нехваткой ресурсов. Также имейте в виду, что если запускать контейнеры на узле, который не управляется Kubernetes, это отразится на вычислениях емкости узла в Kubernetes.

Обойти это ограничение можно, запустив под-заглушку, который ничего не делает, а только имитирует потребление ресурсов в объемах, соответствующих неотслеживаемым контейнерам. Такой под создается только для представления и

ресурсов, потребляемых неотслеживаемыми контейнерами, и чтобы помочь планировщику построить более полную модель ресурсов узла.

Потребности контейнеров в ресурсах

Другим важным условием эффективного размещения подов является учет зависимостей контейнеров от среды времени выполнения и требований к ресурсам. Мы рассмотрели этот вопрос в главе 2 «Предсказуемые требования». Все сводится к тому, что контейнеры должны определять свои профили ресурсов (запросы и лимиты) и зависимости окружения, такие как наличие хранилищ или доступность портов. Только в этом случае поды могут разумно распределяться по узлам и работать, не влияя друг на друга в пиковые периоды.

Политики размещения

Последний элемент — наличие правильных политик фильтрации и приоритетов для конкретных потребностей приложений. Планировщик имеет набор предикатов и политик приоритетов по умолчанию, который подходит для большинства случаев. Его можно переопределить и запустить планировщик с другим набором политик, как показано в листинге 6.2.



Политики и настройки планирования может определить только администратор в ходе конфигурации кластера. Как обычный пользователь вы можете лишь использовать predetermined планировщики.

Листинг 6.2. Пример политики планирования

```
{
  "kind" : "Policy",
  "apiVersion" : "v1",
  "predicates" : [
    {"name" : "PodFitsHostPorts"},
    {"name" : "PodFitsResources"},
    {"name" : "NoDiskConflict"},
    {"name" : "NoVolumeZoneConflict"},
    {"name" : "MatchNodeSelector"},
    {"name" : "HostName"}
  ],
  "priorities" : [
    {"name" : "LeastRequestedPriority",
"weight" : 2},
    {"name" : "BalancedResourceAllocation",
"weight" : 1},
    {"name" : "ServiceSpreadingPriority",
"weight" : 2},
    {"name" : "EqualPriority", "weight" :
1}
  ]
}
```

❶ Предикаты — это правила фильтрации неподходящих узлов. Например, *PodFitsHostsPorts* планирует поды, требующие наличия определенных фиксированных портов, только на узлах, где эти порты еще доступны.

❷ Приоритеты — это правила, которые сортируют доступные узлы в соответствии с предпочтениями. Например, *LeastRequestedPriority* дает узлам с меньшим количеством запрашиваемых ресурсов более высокий приоритет.

Имейте в виду, что, кроме настройки политик планировщика по умолчанию, также можно запустить несколько планировщиков и позволить подам указывать, какой планировщик должен их размещать. Можно запустить другой экземпляр планировщика с другими настройками, дав ему уникальное имя. А затем просто добавить поле `.spec.schedulerName` с именем этого планировщика в настройке пода, после чего этот под будет выбираться для планирования только этим специализированным планировщиком.

Процесс планирования

Поды назначаются узлам, обладающим определенной емкостью, в соответствии с политиками размещения. Для полноты обсуждения на рис. 6.1 в общих чертах показано, как все необходимые элементы собираются вместе и какие основные этапы преодолевает под в процессе планирования.



Рис. 6.1. Процесс выбора узла для пода

Как только появляется под, который пока не назначен никакому узлу, он выбирается планировщиком вместе со всеми узлами, доступными для планирования, и набором политик фильтрации и приоритетов. На первом этапе планировщик применяет политики фильтрации и исключает из дальнейшего рассмотрения все узлы, которые не

соответствуют критериям пода. На втором этапе оставшиеся узлы упорядочиваются по весу. На последнем этапе поду назначается узел, что является главным результатом процесса планирования.

В большинстве случаев лучше позволить планировщику самому назначить узел поду, а не пытаться реализовать свою логику размещения. Однако иногда может потребоваться принудительно связать под с конкретным узлом или группой узлов. Это назначение можно сделать с помощью селектора узла. `.spec.nodeSelector` — это поле в настройках пода, которое определяет массив пар ключ/значение, которые должны присутствовать в виде меток на узлах, пригодных для назначения данному поду. Например, предположим, что некоторый под должен выполняться на определенном узле, где имеется хранилище SSD или поддерживается ускорение вычислений на GPU. Тогда, если добавить в определение пода поле `nodeSelector` с содержимым `disktype: ssd`, как показано в листинге 6.3, только узлы с меткой `disktype = ssd` будут иметь право выполнять этот под.

Листинг 6.3. Селектор узла по типу доступного диска

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
  nodeSelector:
    disktype: ssd
```

❶ Набор меток, которыми должен быть отмечен узел, пригодный для выполнения этого пода.

Помимо определения дополнительных меток узлов, можно использовать некоторые из меток которыми отмечен каждый узел по умолчанию. Например, каждый узел имеет уникальную метку `kubernetes.io/hostname`, которую можно использовать для размещения подов на узлах, исходя из имени хоста. Также можно использовать другие метки по умолчанию, которые определяют операционную систему, аппаратную архитектуру и типы экземпляров.

Близость узлов

Kubernetes поддерживает еще много весьма гибких способов настройки процесса планирования. Один такой способ — определение степени близости узлов, который является более общей формой способа на основе селектора узла, описанного выше, и позволяет задать правила, обязательные или предпочтительные. Обязательные правила должны выполняться всегда, чтобы узел мог быть выбран для запуска пода, тогда как предпочтительные правила подразумевают степень предпочтения, увеличивая вес соответствующих узлов, но не являются обязательными. Кроме того, поддержка понятия близости узлов позволяет выразить весьма широкий спектр ограничений, добавляя такие операторы, как `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt` и `Lt`. В листинге 6.4 показано, как определяется близость узлов.

Листинг 6.4. Определение пода с описанием близости узлов

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
```

```

spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution: ❶
      nodeSelectorTerms:
        -
          matchExpressions: ❷
            - key: numberCores
              operator: Gt
              values: [ "3" ]
            - preferredDuringSchedulingIgnoredDuringExecution: ❸
              - weight: 1
                preference:
                  matchFields: ❹
                    - key: metadata.name
                      operator: NotIn
                      values: [ "master" ]
          containers:
            - image: k8spatterns/random-generator:1.0
              name: random-generator

```

❶ Жесткое требование: узел должен иметь больше трех ядер (обозначается меткой узла), чтобы участвовать в процессе планирования. Правило не пересматривается во время выполнения, если условия на узле изменятся.

❷ Соответствие определяется меткой.

❸ Нежесткое требование: список селекторов с весами. К весу каждого узла прибавляются веса совпавших селекторов,

после чего выбирается узел с самым высоким весом, если он соответствует жестким требованиям.

④ Соответствие определяется полем (задается как `jsonpath`). Обратите внимание, что здесь допускается использовать только операторы `In` и `NotIn`, и в списке значений `values` может присутствовать только одно значение.

Близость и удаленность подов

Близость узлов является мощным инструментом планирования и следует выбирать именно его, когда возможностей селектора узла `nodeSelector` оказывается недостаточно. Этот механизм позволяет ограничить круг узлов, на которых может выполняться под, опираясь на соответствие меток или полей. Но он не позволяет выразить зависимости между подами и указать, как они должны размещаться относительно других подов. Для выражения требований к размещению подов с целью достижения высокой доступности или уменьшения задержек во взаимодействиях можно использовать оценку близости и удаленности подов.

Близость узлов помогает настроить выбор узлов, а близость подов не ограничивается узлами и позволяет выражать правила, затрагивающие несколько уровней топологии. Используя поле `topologyKey` и соответствующие метки, можно применять более детальные правила, которые комбинируют правила выбора домена, стойки, зоны облачного провайдера и региона, как показано в листинге 6.5.

Листинг 6.5. Определение пода с описанием близости узлов

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
```

```

spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution: ❶
-
labelSelector:
❷
  matchLabels:
    confidential: high
    topologyKey: security-
zone ❸
  podAntiAffinity:
    ❹
    preferredDuringSchedulingIgnoredDuringExecution: ❺
    - weight: 100
      podAffinityTerm:
        labelSelector:
          matchLabels:
            confidential: none
            topologyKey: kubernetes.io/hostname
  containers:
    - image: k8spatterns/random-generator:1.0
      name: random-generator

```

❶ Обязательные правила, касающиеся размещения подов с учетом других подов, выполняющихся на целевом узле.

❷ Метка-селектор для поиска подов, размещаемых вместе с данным.

❸ Узлы, на которых выполняются поды с метками `confidential=high`, должны иметь метку `security-zone`.

Для определяемого здесь пода будет выбран узел с той же меткой и значением.

④ Правила удаленности описывают узлы, на которых под *не* может быть размещен.

⑤ Правило говорит, что под *не* должен (но может) размещаться на любых узлах, где выполняются поды с меткой `confidential=none`.

Подобно близости узлов, существуют жесткие и мягкие требования к близости и удаленности подов, которые называются

`requiredDuringSchedulingIgnoredDuringExecution` и `preferredDuringSchedulingIgnoredDuringExecution` соответственно. И так же как в случае близости узлов, в имени поля присутствует окончание `IgnoredDuringExecution`¹³, которое добавлено для возможности расширения в будущем. В настоящий момент, если метки на узле изменятся и правила близости станут недействительными, поды продолжат выполняться¹⁴, но в будущем такие изменения во время выполнения, возможно, будут учитываться.

Непригодность и допустимость

Еще одна возможность, помогающая управлять выбором узлов для выполнения пода, основана на понятиях непригодности и допустимости. Оценка близости узлов позволяет подам выбирать наиболее подходящие узлы, а непригодность и допустимость имеют противоположное назначение. Они позволяют узлам определять, какие поды должны или не должны планироваться на них. *Непригодность* — это характеристика узла, и если она определена, эта характеристика не позволяет планировать поды для выполнения на узле, если они непригодны для этого. В этом

смысле непригодность и допустимость можно рассматривать как *условие включения*, позволяющее планировать на узлах, которые по умолчанию недоступны для планирования, тогда как правила близости являются *условием исключения*, явно определяющим, на каких узлах может выполняться под, и исключаящим все невыбранные узлы.

Непригодность добавляется в узел с помощью `kubectl`:
`kubectl taint nodes master`
`node-role.kubernetes.io/master="true":NoSchedule`,
что оказывает эффект, представленный в листинге 6.6. Соответствующая допустимость добавляется в определение пода, как показано в листинге 6.7. Обратите внимание, что параметры `key` и `effect` в разделе `taints` в листинге 6.6 и в разделе `tolerations` в листинге 6.7 имеют одни и те же значения.

Листинг 6.6. Непригодность узла

```
apiVersion: v1
kind: Node
metadata:
  name: master
spec:
  taints:
    - effect: NoSchedule
      key: node-role.kubernetes.io/master
```

❶

❶ Отметить этот узел как непригодный для планирования, если только под не определяет эту непригодность допустимой.

Листинг 6.7. Определение пригодности недопустимого узла

```
apiVersion: v1
kind: Pod
metadata:
```



```
name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
  tolerations:
  - key: node-role.kubernetes.io/master ❶
    operator: Exists
    effect: NoSchedule ❷
```

❶ Считать допустимыми (доступными для планирования) объявленные непригодными узлы с ключом `node-role.kubernetes.io/master`. В промышленных кластерах этот признак непригодности устанавливается на главном узле для предотвращения планирования подов на нем. Определение допустимости, как в этом поде, позволяет запустить данный под на главном узле.

❷ Считать допустимым, только если определен эффект `NoSchedule`. Это поле может быть пустым, и тогда допустимыми будут считаться любые эффекты.

Условие непригодности может быть жестким, предотвращающим планирование на узле (`effect = NoSchedule`); нежестким, рекомендующим избегать планирования на узле (`effect = PreferNoSchedule`); и вытесняющим уже запущенные поды с узла (`effect = NoExecute`).

Условия непригодности и допустимости помогают определять сложные варианты использования, такие как создание выделенных узлов для ограниченного набора подов, или принудительно вытеснять поды с проблемных узлов, объявляя их непригодными.

Вы можете влиять на размещение подов, чтобы обеспечить высокую доступность и производительность приложения, но старайтесь не сильно ограничивать планировщик и не загоняйте себя в угол, когда планирование новых подов оказывается невозможным, хотя ресурсов более чем достаточно. Например, если требования ваших контейнеров к ресурсам слишком грубые или узлы слишком малы, у вас могут оказаться незаполненными ресурсы на узлах, которые не используются.

На рис. 6.2 можно видеть, что узел А имеет 4 Гбайт незанятой памяти, потому что не осталось свободных ядер процессора для размещения других контейнеров. Создание контейнеров с меньшими требованиями к ресурсам могло бы помочь исправить ситуацию. Другое решение состоит в том, чтобы использовать *механизм перепланирования* в Kubernetes, который помогает дефрагментировать узлы и повысить степень их использования.

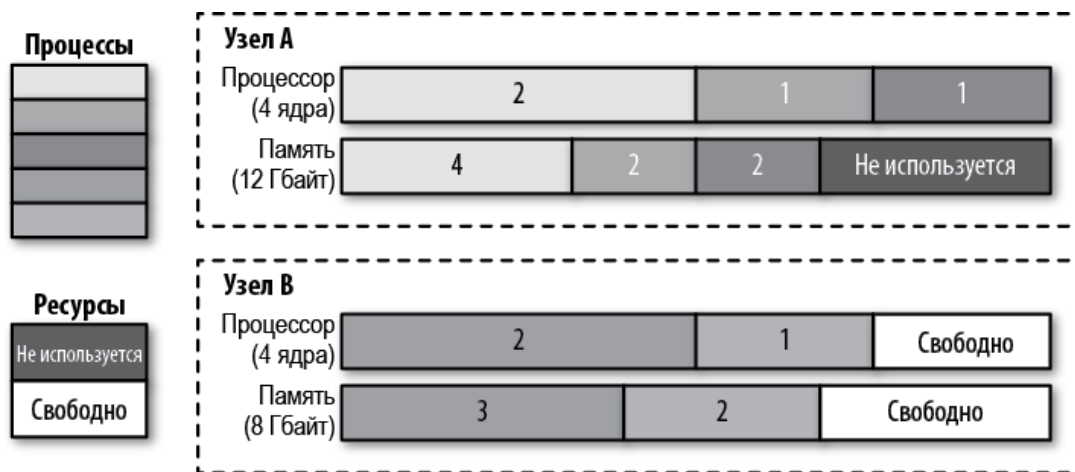


Рис. 6.2. Процессы, запланированные на узлах, и неизрасходованные ресурсы

После запуска пода на узле работа планировщика завершается, и местоположение пода больше не меняется, пока тот не будет удален и воссоздан без назначения узла. Как вы уже видели, со временем это может привести к

фрагментации и недоиспользованию ресурсов кластера. Другая потенциальная проблема заключается в том, что решения планировщика основываются на его представлении о кластере в момент планирования нового пода. Если кластер меняется динамически, или меняются профили ресурсов узлов, или добавляются новые узлы, планировщик не будет перераспределять ранее запущенные поды. Кроме емкости узлов, также могут меняться их метки, влияющие на размещение, но это также никак не влияет на перераспределение уже запущенных подов.

Все эти сценарии относятся к ведению механизма перепланирования. Механизм перепланирования в Kubernetes относится к разряду вспомогательных и обычно запускается как задание, когда администратор кластера решает, что пришло время привести дефрагментацию кластера, перераспределив поды. Механизм перепланирования поддерживает некоторые предопределенные политики, которые можно включать, настраивать или отключать. Политики определяются в файле с настройками пода механизма перепланирования и в настоящее время включают:

RemoveDuplicates

Стратегия удаления дубликатов гарантирует выполнение на одном узле только одного пода, связанного с набором реплик ReplicaSet или развертыванием Deployment. Если обнаружится большее число подов, лишние поды будут вытеснены. Эта стратегия может пригодиться в сценариях, когда узел вышел из строя и управляющие контроллеры запустили новые поды на других, исправных узлах. Когда неисправный узел восстановится и вернется в кластер, количество запущенных модулей окажется больше желаемого, и тогда перепланировщик сможет помочь

вернуть их число к нужному количеству реплик. Удаление дубликатов на узлах также может помочь равномерно распределить поды между несколькими узлами, если политики планирования и топология кластера изменятся после первоначального размещения.

LowNodeUtilization

Эта стратегия выявляет узлы с низким коэффициентом использования ресурсов и переносит на них поды с других, перегруженных узлов, в надежде добиться более удачного распределения и равномерного использования ресурсов. Под узлами с недоиспользованными ресурсами подразумеваются узлы, на которых количество ядер, объем памяти или число подов меньше настроенных пороговых значений `thresholds`. Аналогично, под перегруженными узлами подразумеваются узлы, на которых эти значения превышают настроенные значения `targetThresholds`. Любой узел, использование ресурсов которого находится между этими значениями, считается оптимально используемым и не подвергается действию этой стратегии.

RemovePodsViolatingInterPodAntiAffinity

Эта стратегия исключает блоки, нарушающие правила удаленности между подами, что может произойти при добавлении правил определения удаленности уже после размещения подов на узлах.

RemovePodsViolatingNodeAffinity

Эта стратегия предназначена для вытеснения подов, нарушающих правила близости.

Независимо от используемой политики, перепланировщик старается избежать вытеснения:

- критически важных подов, отмеченных аннотацией `scheduler.alpha.kubernetes.io/critical-pod`;
- подов, не управляемых набором реплик `ReplicaSet`, развертыванием `Deployment` или заданием `Job`;
- подов, управляемых контроллером `DaemonSet`;
- подов, имеющих локальное хранилище;
- подов с параметром `PodDisruptionBudget`, если вытеснение может нарушить установленные правила;
- пода самого перепланировщика (с этой целью под перепланировщика маркируется как критически важный).

Конечно, все операции вытеснения производятся с учетом уровня качества обслуживания подов, то есть первыми вытесняются поды с негарантированным качеством обслуживания, затем поды с переменным качеством обслуживания и, наконец, поды с гарантированным качеством обслуживания. Подробнее об уровнях качества обслуживания рассказывается в главе 2 «Предсказуемые требования».

Пояснение

Размещение — это область, в которую вмешательство нежелательно. Если вы будете следовать рекомендациям из

главы 2 «Предсказуемые требования» и объявите все потребности контейнера в ресурсах, планировщик выполнит свою работу и поместит под на наиболее подходящий узел. Однако если этого недостаточно, у вас в запасе есть несколько способов помочь планировщику прийти к желаемой топологии развертывания. В заключение перечислим подходы к управлению планированием (имейте в виду, что на момент написания этой книги данный список менялся с каждой новой версией Kubernetes), в порядке возрастания их сложности:

nodeName

Простейшая форма связывания пода с узлом. В идеале это поле должно заполняться планировщиком, который руководствуется политиками, а не вручную. Связывание пода с конкретным узлом значительно сужает область для планирования этого пода. Это возвращает нас к эпохе, предшествовавшей Kubernetes, когда мы явно указывали узлы для запуска наших приложений.

nodeSelector

Массив пар ключ/значение. Чтобы под можно было запустить на узле, указанные пары ключ/значение должны присутствовать на нем в виде меток. Селектор узла — один из самых простых механизмов управления выбором планировщика, требующий всего лишь добавить несколько значимых меток в определение пода и узла (что приходится делать в любом случае).

Изменение правил планирования по умолчанию

Планировщик по умолчанию отвечает за размещение новых подов на узлах в кластере и справляется с этой задачей достаточно хорошо. Однако при необходимости можно изменить список политик фильтрации и приоритетов планировщика.

Близость и удаленность пода

Эти правила позволяют выражать зависимости одних подов от других, например, для уменьшения задержек в приложении, увеличения доступности, удовлетворения ограничений безопасности и т.д.

Близость узла

Это правило позволяет выразить зависимость пода от узлов. Например, учесть наличие оборудования, местоположение и т.д.

Непригодность и допустимость

Непригодность и допустимость позволяют узлу контролировать, какие поды могут или не могут планироваться на них, например, чтобы выделить узел для группы подов или даже вытеснить поды во время выполнения. Другое преимущество настроек непригодности и допустимости состоит в том, что при расширении кластера Kubernetes путем добавления новых узлов с новыми метками вам не придется добавлять новые метки во все поды — это нужно будет сделать только для подов, которые должны быть размещены на новых узлах.

Нестандартный планировщик

Если ни один из предыдущих подходов не является достаточно хорошим или ваши требования к планированию слишком сложны, вы можете реализовать свой планировщик. Такой планировщик может работать вместо стандартного планировщика Kubernetes или одновременно с ним. Можно использовать гибридный подход, когда запускается процесс «расширения планировщика», к которому обращается стандартный планировщик Kubernetes на последнем этапе перед принятием решения о планировании. При таком подходе не требуется реализовывать полный планировщик и достаточно лишь предоставить HTTP API для фильтрации и определения приоритетов узлов. Преимущество создания своего планировщика состоит в том, что при этом можно учитывать факторы, внешние по отношению к кластеру Kubernetes, такие как стоимость оборудования, сетевые задержки и оптимизация использования ресурсов при распределении подов между узлами. Также есть возможность задействовать несколько своих планировщиков вместе с планировщиком по умолчанию и выбирать, какой планировщик использовать для каждого пода. Каждый планировщик может иметь свой набор политик для своего подмножества подов.

Как видите, есть много способов управлять размещением подов, и выбор правильного подхода или комбинирование нескольких подходов могут оказаться сложной задачей. Основная мысль этой главы: определите и объявите профили ресурсов контейнера, снабдите поды и узлы соответствующими метками, наконец, старайтесь поменьше вмешиваться в работу планировщика Kubernetes.

Дополнительная информация

- Пример автоматизированного размещения (<http://bit.ly/2TTJUMh>).
- Связывание подов с узлами (<https://kubernetes.io/docs/user-guide/node-selection/>).
- Описание планирования и размещения узлов (<https://red.ht/2TP1ceB>).
- Бюджет неработоспособности пода (<https://kubernetes.io/docs/admin/disruptions/>).
- Гарантии планирования критически важных подов (<https://kubernetes.io/docs/admin/rescheduler/>).
- Планировщик Kubernetes (<http://bit.ly/2Hrq8U>).
- Алгоритм планирования (<http://bit.ly/2F9Vfi2>).
- Настройка нескольких планировщиков (<http://bit.ly/2HLv5Fk>).
- Перепланировщик для Kubernetes (<http://bit.ly/2YMQzYn>).
- Статья «Keep Your Kubernetes Cluster Balanced: The Secret to High Availability» (<http://bit.ly/2zuecKk>).
- Видеоролик «Everything You Ever Wanted to Know About Resource Scheduling, but Were Afraid to Ask» (<http://bit.ly/2FNkBT9>).

[14](#) Если метки узла изменятся и поды, прежде несовместимые с этим узлом в соответствии с селектором близости, окажутся совместимыми, они будут планироваться для выполнения на этом узле.

Часть II. Поведенческие паттерны

Паттерны этой категории предназначены для создания коммуникационных механизмов и организации взаимодействий между подами и управляющей платформой. В зависимости от типа управляющего контроллера под может выполняться до завершения или запускаться и останавливаться по расписанию. Он может работать как *фоновая служба* или предоставлять гарантии уникальности своим репликам. Существуют различные способы запуска подов, и выбор правильных средств управления подами требует понимания их поведения. В следующих главах мы рассмотрим паттерны:

- Глава 7 «Пакетное задание» описывает изолированную и атомарную единицу работы, которая выполняется до своего завершения.
- Глава 8 «Периодическое задание» описывает паттерн, который позволяет запускать единицу работы по временным событиям.
- Глава 9 «Фоновая служба» описывает паттерн, который позволяет запускать поды поддержки инфраструктуры на определенных узлах до размещения прикладных подов.
- Глава 10 «Служба-одиночка» описывает паттерн, который гарантирует наличие единственного активного экземпляра службы в каждый момент времени и его высокую доступность.
- Глава 11 «Служба с состоянием» описывает паттерн создания распределенных приложений с поддержкой сохранения

своего состояния и управления ими с помощью Kubernetes.

- Глава 12 «Обнаружение служб» описывает механизмы, с помощью которых клиенты могут обнаруживать экземпляры служб, предлагаемых приложением, и обращаться к ним.
- Глава 13 «Самоанализ» описывает механизмы интроспекции и внедрения метаданных в приложения.

Глава 7. Пакетное задание

Паттерн *Batch Job* (Пакетное задание) предназначен для управления изолированными атомарными единицами работы. Он основан на абстракции задания *Job*, предназначенной для запуска короткоживущих подов, действующих в распределенной среде и завершающихся самостоятельно.

Задача

Основным примитивом для запуска контейнеров и управления ими является под (*Pod*) — группа контейнеров. Существуют разные способы создания подов с различными характеристиками:

Простой под

Позволяет вручную создать под для запуска контейнеров. Однако когда узел, на котором выполняется такой под, выходит из строя, под не перезапускается. Запускать поды таким способом не рекомендуется, за исключением случаев разработки или тестирования. Этот механизм также известен под названиями *неуправляемые*, или *голые*, поды.

Набор реплик ReplicaSet

Этот контроллер используется для создания и управления жизненным циклом подов, предназначенных для непрерывного выполнения (например, для запуска контейнера веб-сервера). Он поддерживает стабильный набор реплик подов, действующих непрерывно, и

гарантирует наличие определенного количества идентичных подов.

Набор демонов DaemonSet

Контроллер для запуска единственного пода на каждом узле. Обычно используется для управления механизмами платформы, такими как мониторинг, объединение журналов, хранилища и др. Подробнее о наборах демонов рассказывается в главе 9 «Фоновая служба».

Объединяет все эти поды тот факт, что они являются процессами, которые действуют продолжительное время и которые не должны останавливаться. Однако иногда бывает желательно выполнить предопределенную конечную единицу работы и затем остановить контейнер. Для этой цели Kubernetes предоставляет ресурс задания Job.

Решение

Задание Job в Kubernetes напоминает набор реплик ReplicaSet — оно точно так же создает один или несколько подов и обеспечивает их выполнение. Однако, в отличие от набора реплик, после успешного завершения ожидаемого количества подов задание считается выполненным и дополнительные поды не запускаются. В листинге 7.1 показано, как выглядит определение задания.

Листинг 7.1. Определение задания Job

```
apiVersion: batch/v1
kind: Job
metadata:
  name: random-generator
spec:
```

```

completions: 5           ❶
parallelism: 2          ❷
template:
  metadata:
    name: random-generator
  spec:
    restartPolicy: OnFailure  ❸
    containers:
      - image: k8spatterns/random-generator:1.0
        name: random-generator
          command: [ "java", "-cp", "/",
"RandomRunner",
                    "/numbers.txt", "10000" ]

```

❶ Задание должно выполнить пять подов, и все они должны завершиться с признаком успеха.

❷ Одновременно могут выполняться два пода.

❸ В определениях заданий требуется обязательно задать параметр `restartPolicy`.

Одним из важных различий между заданием Job и определением набором реплик ReplicaSet является определение параметра `.spec.template.spec.restartPolicy`. Для набора реплик этот параметр получает значение по умолчанию `Always` (всегда), имеющее смысл для длительных процессов, которые должны выполняться постоянно. Значение `Always` недопустимо для заданий, и единственными возможными вариантами являются `OnFailure` (в случае сбоя) или `Never` (никогда).

Итак, почему для однократного запуска лучше создавать задание вместо простого пода? Задания дают много

преимуществ в смысле надежности и масштабируемости, что делает их предпочтительным вариантом:

- Задание Job — это не эфемерный процесс в памяти, а постоянная задача, которая автоматически восстанавливается после перезапуска кластера.
- По завершении задание не удаляется, а сохраняется и доступно для трассировки. Поды, созданные как часть задания, также не удаляются и доступны для исследования (например, для исследования журналов контейнера). То же верно и для простых подов, но только если установлен параметр `restartPolicy: OnFailure`.
- Задание можно выполнить несколько раз. В параметре `.spec.completions` можно указать, сколько раз под должен успешно завершиться, чтобы задание было признано выполненным.
- Когда задание должно выполниться несколько раз (согласно значению параметра `.spec.completions`), его также можно масштабировать и одновременно запускать несколько подов. Это можно сделать, определив параметр `.spec.parallelism`.
- Если узел выходит из строя или по какой-то причине вытесняется во время работы, планировщик разместит и запустит под на новом исправном узле. Простые поды будут оставаться в неисправном состоянии, поскольку такие поды никогда не перемещаются на другие узлы автоматически.

Все это делает механизм заданий привлекательным для использования в сценариях, когда необходимы гарантии

выполнения единицы работы.

Главные роли в поведении задания играют два параметра:

.spec.completions

Определяет, сколько раз следует запустить под, чтобы задание считалось выполненным.

.spec.parallelism

Определяет, сколько реплик пода может выполняться параллельно. Большое число в этом параметре не гарантирует высокого уровня параллелизма, и на самом деле количество одновременно выполняемых подов может быть меньше (а в некоторых тупиковых ситуациях больше) желаемого (например, из-за особенностей регулирования, ограниченности ресурсов, при приближении к числу `.spec.completions` и по другим причинам). Запись значения 0 в этот параметр эффективно приостанавливает задание.

На рис. 7.1 показано, как действует паттерн *Batch Job* (Пакетное задание), настройки которого приводились в листинге 7.1, со счетчиком выполнений, равным пяти, и уровнем параллелизма, равным двум.

Исходя из значений этих двух параметров, можно выделить следующие типы заданий:

Задание с одиночным подом

Этот тип выбирается, когда вы оставляете значение по умолчанию 1 в обоих параметрах, `.spec.completions` и `.spec.parallelism`. Это задание

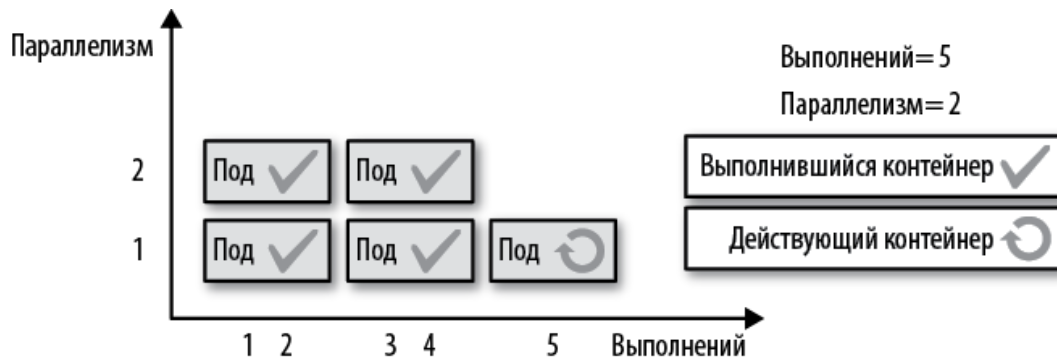


Рис. 7.1. Параллельное выполнение заданий в паттерне Batch Job с фиксированным счетчиком выполнений

запускает только один под и завершается по окончании выполнения этого единственного пода с признаком успеха (с кодом 0).

Задание с фиксированным числом выполнений

Число в параметре `.spec.completions` определяет, сколько раз должен выполняться под и завершиться с признаком успеха. При желании можно также установить параметр `.spec.parallelism` или оставить в нем значение по умолчанию, равное единице. Такое задание считается выполненным после того, как под успешно завершится `.spec.completions` раз. Листинг 7.1 демонстрирует, как действует этот режим, который считается лучшим выбором, когда количество единиц работы известно заранее и стоимость обработки одной единицы оправдывает использование выделенного пода.

Рабочая очередь

Опустив параметр `.spec.completions` и присвоив параметру `.spec.parallelism` целое число больше единицы, вы получите рабочую очередь для параллельных

заданий. Задание типа «рабочая очередь» считается выполненным, если все поды завершились и хотя бы один из них завершился с признаком успеха. В этом сценарии требуется, чтобы поды координировали свою работу друг с другом и определяли, над чем каждый будет работать. Например, когда в очереди имеется фиксированное, но неизвестное количество элементов для обработки, параллельно выполняющиеся поды могут выбирать их один за другим и обрабатывать. Первый под, обнаруживший, что очередь опустела, и завершившийся с признаком успеха, указывает на выполнение задания. Контроллер Job также ожидает завершения всех других подов. Поскольку один под может обработать несколько элементов, этот тип заданий является отличным выбором, когда накладные расходы на запуск одного пода для обработки одного элемента оказываются неоправданно высокими.

Для обработки неограниченного потока элементов лучше использовать другие контроллеры управления подами, такие как набор реплик ReplicaSet.

Пояснение

Абстракция задания Job — простой, но важный примитив, на котором основываются другие примитивы, такие как планировщик заданий CronJob. Задания помогают превратить изолированные единицы работы в надежную и масштабируемую единицу выполнения. Однако задание никак не определяет, как отдельные элементы для обработки должны отображаться в задания или поды. Это вы должны сделать сами, приняв во внимание плюсы и минусы каждого варианта:

Отдельное задание Job для каждой единицы работы

Этот вариант влечет дополнительные накладные расходы на создание заданий Kubernetes, а также на управление большим количеством заданий, которые потребляют ресурсы кластера. Этот вариант может пригодиться, когда каждая единица работы является сложной задачей и ее необходимо фиксировать, исследовать или масштабировать независимо.

Одно задание на все единицы работы

Этот вариант подходит для сценариев, когда имеется большое количество единиц работы, которые не требуется фиксировать, исследовать или масштабировать независимо. В этом сценарии управление выполнением единиц работы может осуществляться средствами приложения.

Контроллер заданий Job реализует лишь самый минимум, необходимый для планирования единиц работы. Любая более сложная реализация должна объединять примитив Job с фреймворками пакетной обработки (такими, как Spring Batch и JBeret в экосистеме Java) для достижения желаемого результата.

Не все службы должны работать постоянно. Некоторые могут выполняться по требованию, некоторые – в определенные моменты времени, а некоторые – периодически. Использование заданий позволяет запускать поды только при необходимости и только на время выполнения задачи. Задания планируются на узлах, которые имеют необходимые ресурсы, удовлетворяют политикам размещения подов и другим зависимостям контейнера. Использование заданий для кратковременных задач вместо долгоживущих абстракций (таких, как ReplicaSet) экономит ресурсы платформы. Все это делает задания Job уникальным

примитивом и помогает платформе Kubernetes поддерживать разные виды рабочей нагрузки.

Дополнительная информация

- Пример пакетного задания (<http://bit.ly/2Jnloz6>).
- Задания Job как средство выполнения единиц работы (<http://bit.ly/2W1ZTW2>).
- Параллельная обработка с использованием заданий (<http://bit.ly/2Y563GL>).
- Грубая параллельная обработка с использованием рабочей очереди (<http://bit.ly/2Y29cqS>).
- Тонкая параллельная обработка с использованием рабочей очереди (<http://bit.ly/2Obtutr>).
- Создание индексированного задания с помощью метаконтроллера (<http://bit.ly/2FkjQSA>).
- Фреймворки и библиотеки Java для пакетной обработки (<https://github.com/jberet>).

Глава 8. Периодическое задание

Паттерн *Periodic Job* (Периодическое задание) расширяет паттерн *Batch Job* (Пакетное задание), добавляя измерение времени и позволяя выполнять единицу работы по временным событиям.

Задача

В мире распределенных систем и микросервисов наблюдается явное стремление к организации взаимодействий и обмену событиями между приложениями в режиме реального времени с использованием HTTP и упрощенных механизмов обмена сообщениями. Однако, независимо от последних тенденций в разработке программного обеспечения, планирование заданий имеет долгую историю и продолжает оставаться актуальным. Паттерн *Periodic Job* (Периодическое задание) обычно используется для автоматизации обслуживания системы и решения административных задач. Он также применяется в бизнес-приложениях, требующих периодического выполнения определенных задач. Типичными примерами могут служить интеграция между системами посредством передачи файлов, интеграция приложений посредством периодического опроса баз данных, отправка электронных писем с новостями, а также архивация и удаление старых файлов.

Традиционно для *периодического выполнения заданий* с целью обслуживания системы используется специализированное программное обеспечение — планировщик Cron. Однако специализированное программное обеспечение может оказаться избыточным для простых случаев использования, к тому же задания Cron, выполняемые

на единственном сервере, трудно поддерживать, так как они представляют собой единую точку отказа. Вот почему очень часто разработчики стремятся реализовать решения, которые способны осуществлять планирование и выполнять необходимую бизнес-логику. Например, в мире Java выполнение заданий с привязкой ко времени можно организовать с использованием библиотек, таких как Quartz и Spring Batch, или пользовательских реализаций на основе класса `ScheduledThreadPoolExecutor`. Но, как и в случае с Cron, основная сложность этого подхода состоит в том, чтобы сделать механизм планирования устойчивым и высокодоступным, а это влечет значительное потребление ресурсов. Кроме того, при таком подходе планировщик заданий является частью приложения, и чтобы обеспечить высокую доступность планировщика, нужно обеспечить высокую доступность всего приложения. Обычно ради этого приходится запускать несколько экземпляров приложения, но так, чтобы активно занимался планированием только один экземпляр, а это требует реализации алгоритма выбора лидера и решения других проблем, характерных для распределенных систем.

В конце концов, простая служба, которая должна скопировать несколько файлов один раз в день, может потребовать нескольких узлов, распределенного механизма выбора лидера и многого другого. Реализация контроллера `CronJob` в Kubernetes решает все эти проблемы и позволяет планировать ресурсы `Job` с помощью хорошо известного формата описания заданий, используемого планировщиком Cron. Это дает возможность разработчикам сосредоточиться только на реализации выполняемой работы, а не на аспектах планирования с привязкой ко времени.

Решение

В главе 7 «Пакетное задание» мы познакомились с возможностями и вариантами использования поддержки заданий Job в Kubernetes. Все, о чем рассказывалось там, относится и к этой главе, потому что примитив CronJob основан на Job. Экземпляр CronJob напоминает строку в файле *crontab* в Unix (таблица заданий планировщика cron) и управляет аспектами выполнения задания, связанными со временем. Он позволяет периодически выполнять задание в определенные моменты времени. Образец определения такого периодического задания приводится в листинге 8.1.

Листинг 8.1. Ресурс CronJob

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: random-generator
spec:
  # Через каждые три минуты
  schedule: "* /3 * * * *"           ❶
  jobTemplate:
    spec:
      template:                       ❷
        spec:
          containers:
            - image: k8spatterns/random-
generator:1.0
              name: random-generator
              command: [ "java", "-cp", "/",
"RandomRunner",
"/numbers.txt", "10000"
]
```


restartPolicy: OnFailure

❶ Инструкция для Cron, требующая запускать задание каждые три минуты.

❷ Паттерн задания, имеет ту же структуру, что и описание обычного задания Job.

Кроме описания задания Job, в определении CronJob имеются дополнительные параметры для определения привязки ко времени:

.spec.schedule

Строка в формате *crontab*, определяющая планирование задания Job с привязкой ко времени (например, "0 * * * *" требует запускать задание в начале каждого часа).

.spec.startingDeadlineSeconds

Крайний срок (в секундах) запуска задания, если было пропущено запланированное время. Некоторые задания допустимо выполнять только в течение определенного отрезка времени и бессмысленно запускать позднее. Например, если задание не было запущено в нужное время из-за недостатка вычислительных ресурсов или отсутствия других условий, иногда лучше пропустить выполнение, потому что данные, которые должно обработать это задание, уже устарели.

.spec.concurrencyPolicy

Определяет порядок управления одновременным выполнением заданий, созданных одним и тем же CronJob.

Значение по умолчанию `Allow` (разрешить) позволяет запускать новые экземпляры заданий, даже если предыдущие задания еще не завершились. Если такое поведение нежелательно, можно указать в этом параметре значение `Forbid` (запретить), чтобы пропустить запуск нового экземпляра задания, или значение `Replace` (заменить), чтобы остановить прежний экземпляр задания и запустить новый.

.spec.suspend

Этот параметр приостанавливает запуск новых экземпляров задания, но не влияет на уже запущенные.

.spec.successfulJobsHistoryLimit

.spec.failedJobsHistoryLimit

Эти поля определяют, сколько заданий, выполнившихся успешно и завершившихся с ошибкой, следует сохранить для исследования.

`CronJob` — это узкоспециализированный примитив и применяется, только когда единица работы имеет временное измерение. Даже при том, что `CronJob` не является примитивом общего назначения, он служит отличным примером того, как механизмы Kubernetes основываются друг на друге и поддерживают сценарии использования, не связанные с облачными вычислениями.

Пояснение

Как видите, `CronJob` — это довольно простой примитив, добавляющий кластерное Cron-подобное поведение к

существующему механизму заданий Job. Но когда в сочетании с другими примитивами, такими как Pod, средствами изоляции ресурсов контейнера и другими особенностями Kubernetes, например, описанными в главе 6 «Автоматическое размещение» или в главе 4 «Проверка работоспособности», он превращается в очень мощную систему планирования заданий. Это позволяет разработчикам сосредоточиться исключительно на предметной области и заняться реализацией контейнерного приложения, отвечающей только за выполнение бизнес-логики. Планирование осуществляется за рамками приложения и является частью платформы со всеми ее дополнительными преимуществами, такими как высокая доступность, отказоустойчивость, емкость и размещение подов на основе политик. Конечно, по аналогии с реализацией заданий Job, при реализации контейнера для выполнения под управлением CronJob следует учитывать все тупиковые ситуации: повторные запуски, пропуск запусков, параллельные запуски или принудительная остановка.

Дополнительная информация

- Пример периодически выполняемого задания (<http://bit.ly/2HGXAnh>).
- Описание механизма CronJob (<https://kubernetes.io/docs/concepts/jobs/cron-jobs/>).
- Описание планировщика cron в Unix (<https://ru.wikipedia.org/wiki/Cron>).

Глава 9. Фоновая служба

Паттерн *Daemon Service* (Фоновая служба) позволяет размещать и запускать приоритетные инфраструктурные поды на целевых узлах. Он используется главным образом администраторами для запуска подов, привязанных к конкретным узлам, в целях расширения возможностей платформы Kubernetes.

Задача

Понятие демона (фонового процесса) в программных системах существует на многих уровнях. На уровне операционной системы *демон* — это долго выполняющаяся, самовосстанавливающаяся компьютерная программа, которая запускается как фоновый процесс. В Unix имена демонов обычно заканчиваются на «d», например `httpd`, `named` и `sshd`. В других операционных системах используются альтернативные термины, такие как служебные задачи и фантомные задания.

Независимо от названия, эти программы объединяет то, что они выполняются как процессы, обычно не взаимодействуют с монитором, клавиатурой и мышью и запускаются во время загрузки системы. Аналогичное понятие существует и на уровне приложений. Например, в JVM потоки-демоны работают в фоновом режиме и предоставляют вспомогательные услуги пользовательским потокам выполнения. Эти потоки-демоны имеют низкий приоритет, работают в фоновом режиме, не влияют на жизненный цикл приложения и выполняют такие задачи, как сборка мусора или финализация.

В Kubernetes тоже есть похожее понятие — набор демонов DaemonSet. Как мы знаем, Kubernetes является распределенной платформой, разбросанной по нескольким узлам, основная задача которой — управление подами приложений, поэтому DaemonSet представлен подами, выполняющимися в фоновом режиме на узлах кластера и предоставляющими некоторые услуги остальной части кластера.

Решение

Набор реплик ReplicaSet и его предшественник ReplicationController — это управляющие структуры, отвечающие за выполнение определенного количества подов. Эти контроллеры постоянно проверяют список запущенных подов и следят за тем, чтобы фактическое количество выполняющихся подов всегда соответствовало желаемому. В этом смысле набор демонов DaemonSet действует аналогично и следит, чтобы всегда выполнялось определенное количество подов. Разница лишь в том, что первые два, ReplicaSet и ReplicationController, поддерживают выполнение определенного количества подов, руководствуясь обычными требованиями к высокой доступности и уровню нагрузки для приложений, независимо от количества узлов.

DaemonSet, напротив, решая, сколько экземпляров подов запустить, не учитывает уровень нагрузки. Его главная задача — поддерживать выполнение одного пода на каждом узле или на определенных узлах. Давайте посмотрим, как определяется DaemonSet (листинг 9.1).

Листинг 9.1. Ресурс DaemonSet

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
```

```

name: random-refresher
spec:
  selector:
    matchLabels:
      app: random-refresher
  template:
    metadata:
      labels:
        app: random-refresher
    spec:
      nodeSelector:
        feature: hw-rng
      containers:
        - image: k8spatterns/random-generator:1.0
          name: random-generator
          command:
            - sh
            - -c
            - >-
              "while true; do
                java -cp / RandomRunner
                /host_dev/random 100000;
                sleep 30; done"
          volumeMounts:
            - mountPath: /host_dev
              name: devices
          volumes:
            - name: devices
              hostPath:
                path: /dev

```

❶

❷

❸

❶ Использовать только узлы с меткой `feature` и значением `hw-rng`.

❷ Наборы демонов `DaemonSet` часто монтируют разделы файловой системы узла для решения задач обслуживания.

❸ `hostPath` для прямого доступа к каталогам узла.

При таком поведении основными кандидатами для включения в набор демонов `DaemonSet` обычно являются процессы, связанные с поддержкой инфраструктуры, такие как сборщики журналов, экспортеры метрик и даже маршрутизаторы `kube-proxy`, которые выполняют операции на уровне кластера. Существует много различий в том, как `DaemonSet` и `ReplicaSet` управляют подами, но основные из них следующие:

- По умолчанию `DaemonSet` размещает на каждом узле по одному экземпляру пода. Это поведение можно изменить, определив ограниченное подмножество узлов в параметре `nodeSelector`.
- `DaemonSet` создает под с требуемым значением в параметре `nodeName`. Поэтому `DaemonSet` не требует наличия планировщика Kubernetes для запуска контейнеров. Эта особенность позволяет использовать `DaemonSet` для запуска компонентов Kubernetes и управления ими.
- Поды, созданные контроллером `DaemonSet`, могут запускаться до запуска планировщика, что позволяет запускать их на узлах раньше любых других подов.
- Так как планировщик не используется, контроллер `DaemonSet` не принимает во внимание параметр `unschedulable` узла.

- Поды, управляемые контроллером DaemonSet, как предполагается, должны выполняться только на целевых узлах. По этой причине многие контроллеры обрабатывают их иначе и с более высоким приоритетом. Например, перепланировщик будет избегать вытеснения таких модулей, механизм автоматического масштабирования кластера будет управлять ими отдельно и т.д.

Обычно DaemonSet создает один экземпляр пода на каждом узле или подмножестве узлов. Учитывая это, есть несколько способов достичь подов, управляемых контроллером DaemonSet:

Через службу Service

Создать службу Service с тем же селектором пода, как в DaemonSet, и использовать службу для достижения фонового пода на случайном узле, выбранном балансировщиком нагрузки.

Через DNS

Создать автономную (headless) службу Service с тем же селектором пода, как в DaemonSet, и использовать ее для извлечения нескольких записей A из DNS, содержащих IP-адреса и порты всех подов.

Через NodeIP и hostPort

Поды в DaemonSet могут определять параметр hostPort и становиться доступными через IP-адреса узлов и указанный номер порта. Поскольку комбинация hostIP, hostPort и

protocol должна быть уникальной, количество мест, где под может быть запланирован, ограничено.

Через промежуточное хранилище данных

Приложение в поде из DaemonSet может записывать данные в хорошо известное место или передавать внешней для него службе. В таком случае потребителям не нужно напрямую обращаться к подам в DaemonSet.

статические поды

Другой способ запуска контейнеров подобно тому, как это делает DaemonSet, – использовать *статические поды*. Kubelet, помимо взаимодействия с Kubernetes API Server и получения определений подов, может также получать объявления ресурсов из локального каталога. Подами, которые определены таким способом, управляет только Kubelet, и они выполняются только на одном узле. Программный интерфейс служб не наблюдает за этими подами, они не управляются никакими контроллерами и их работоспособность не проверяется. За такими подами наблюдает только клиент Kubelet, который перезапускает их, когда они завершаются. Аналогично, Kubelet периодически сканирует указанный в настройках каталог, обнаруживает изменения в определениях подов и добавляет или удаляет поды при необходимости.

Статические поды можно использовать для запуска контейнерных версий системных процессов Kubernetes или

других контейнеров. Но контроллеры DaemonSet лучше интегрированы с остальной частью платформы и выглядят предпочтительнее статических подов.

Пояснение

В этой книге мы описываем особенности и паттерны Kubernetes, которые используются в основном разработчиками, а не администраторами. DaemonSet находится где-то посередине, больше тяготея к набору инструментов администратора, но мы включили его в обсуждение, потому что он также часто используется разработчиками приложений. Контроллеры DaemonSet и CronJob также являются прекрасными примерами, как Kubernetes превращает понятия, характерные для одного узла, такие как Crontab и демоны, в кластерные примитивы для управления распределенными системами. Все это новые распределенные понятия, которые разработчики тоже должны знать.

Дополнительная информация

- Пример фоновой службы (<http://bit.ly/2TMX3rc>).
- Описание контроллера DaemonSet (<http://bit.ly/2r07CWx>).
- Выполнение непрерывного обновления в DaemonSet (<http://bit.ly/2CAZ13F>).
- Наборы демонов DaemonSet и задания Job (<http://bit.ly/2HLeHof>).
- Статические поды (<https://kubernetes.io/docs/tasks/administer-cluster/static-pod/>).

Глава 10. Служба-одиночка

Паттерн *Singleton Service* (Служба-одиночка) гарантирует, что в каждый конкретный момент времени активным будет только один экземпляр приложения при условии сохранения высокой доступности. Этот паттерн можно реализовать внутри приложения или полностью переложить на плечи Kubernetes.

Задача

Одна из основных возможностей фреймворка Kubernetes — простота и прозрачность масштабирования приложений. Поды могут масштабироваться императивно, командой `kubectl scale`, или декларативно, путем определения такого контроллера, как `ReplicaSet`, и даже динамически, основываясь на нагрузке на приложение, как описано в главе 24 «Эластичное масштабирование». Запуская несколько экземпляров одной и той же службы (не контроллера службы `Service`, а компонента распределенного приложения, представленного подом), система может увеличивать пропускную способность и доступность. Доступность увеличивается, потому что в случае выхода из строя одного экземпляра службы диспетчер сможет переадресовать запросы другим, исправным экземплярам. В Kubernetes множественные экземпляры являются репликами пода, а ресурс `Service` отвечает за диспетчеризацию запросов.

Однако иногда желательно, чтобы действовал только один экземпляр службы. Например, если служба выполняет некоторую периодическую задачу, тогда при одновременном выполнении нескольких экземпляров каждый будет запускать задачу через запланированные интервалы, что приведет к дублированию, а не к запуску только одной задачи, как

ожидалось. Другой пример: служба, которая выполняет опрос определенных ресурсов (файловой системы или базы данных), и требуется, чтобы такой опрос и обработку результатов осуществлял только один экземпляр и, возможно, даже один поток. Третий случай имеет место, когда речь заходит об однопоточном потребителе, извлекающем сообщения из брокера сообщений, который также является службой-одиночкой.

Во всех этих и подобных им ситуациях нужно иметь возможность ограничивать количество активных экземпляров службы в каждый момент времени (обычно требуется только один), независимо от того, сколько экземпляров было запущено и продолжает работать.

Решение

Запуск нескольких реплик одного модуля создает топологию *активный-активный*, в которой все экземпляры службы активны. Но нам нужна топология *активный-пассивный* (или *ведущий-ведомый*), в которой активен только один экземпляр, а все остальные являются пассивными. Блокировку приложения можно организовать на двух уровнях: извне и изнутри.

Блокировка приложения извне

Как следует из названия, этот подход основан на использовании управляющего процесса, который выполняется отдельно от приложения и обеспечивает работу только одного его экземпляра. Сама реализация приложения не знает об этом ограничении и запускается как экземпляр-одиночка. Это напоминает создание единственного экземпляра класса Java управляющей средой выполнения (например, Spring Framework). Реализация класса не знает, что используется

только один его экземпляр, и не содержит никакого кода, предотвращающего создание нескольких экземпляров.

На рис. 10.1 показано, как можно реализовать блокировку приложения за его пределами с помощью контроллера StatefulSet или ReplicaSet с единственной репликой.

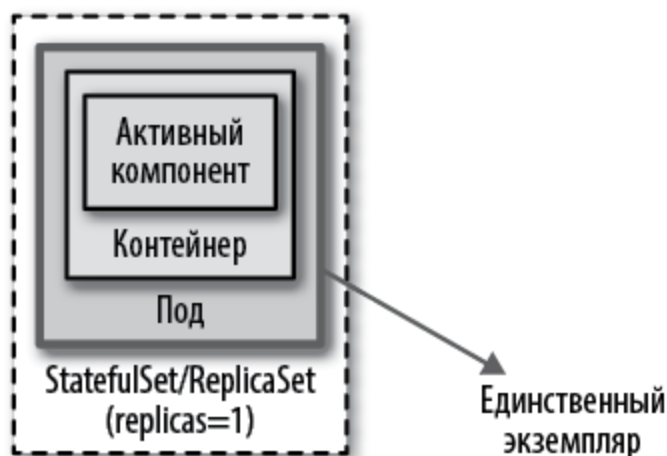


Рис. 10.1. Механизм блокировки приложения извне

В Kubernetes для этого нужно запустить под с единственной репликой. Однако этого недостаточно, чтобы гарантировать высокую доступность пода-одиночки. Дополнительно нужно заключить под в контроллер, такой как ReplicaSet, который обеспечит высокую доступность пода-одиночки. Эта топология не совсем *активный-пассивный* (в данном случае отсутствует пассивный экземпляр), но имеет тот же эффект, так как Kubernetes гарантирует постоянное присутствие в кластере одного действующего экземпляра пода. Кроме того, благодаря контроллеру, осуществляющему проверку работоспособности и повторный запуск пода в случае сбоя, как описано в главе 4 «Проверка работоспособности», обеспечивается высокая доступность единственного экземпляра пода.

Главное, на что следует обратить внимание при реализации этого подхода, — количество реплик, которое не должно

увеличиваться случайно или по ошибке, потому что на уровне платформы отсутствует механизм, предотвращающий изменение количества реплик.

Было бы не совсем верно утверждать, что с этой реализацией в каждый момент времени выполняется только один экземпляр, особенно когда что-то идет не так. Прimitives Kubernetes, такие как ReplicaSet, отдают предпочтение доступности перед согласованностью — осознанное решение для высокодоступных и масштабируемых распределенных систем. Это означает, что ReplicaSet реализует для своих реплик семантику «не менее», а не «не более». Если настроить контроллер ReplicaSet как управляющий одиночным экземпляром, определив параметр `replicas: 1`, он будет гарантировать выполнение не менее одного экземпляра, но иногда может выполняться больше экземпляров.

Чаще других встречается проблема, когда узел с подом, управляемым контроллером, выходит из строя и отключается от кластера Kubernetes. В этом случае контроллер ReplicaSet запускает другой экземпляр пода на исправном узле (при наличии достаточного объема ресурсов), но не гарантирует остановку пода на отключенном узле. Аналогично, при изменении количества реплик или перемещении подов на другие узлы их фактическое количество может временно превысить желаемое. Это временное увеличение производится с целью обеспечить высокую доступность и предотвратить сбои, что необходимо для масштабируемых приложений и приложений без состояния.

Поды-одиночки могут быть отказоустойчивыми и восстанавливаться автоматически, но они не являются высокодоступными по определению. Для одиночек обычно важнее согласованность, а не высокая доступность. В Kubernetes имеется ресурс StatefulSet, который обеспечивает

согласованность в ущерб доступности и дает желаемую гарантию присутствия единственного экземпляра. Если ReplicaSet не предоставляет гарантий, необходимых вашему приложению, и у вас есть строгое ограничение, требующее, чтобы одновременно выполнялось не более одного экземпляра пода, возможно, вам подойдет StatefulSet. Контроллеры StatefulSet предназначены для приложений с состоянием и предлагают много новых возможностей, в том числе более строгие гарантии для одиночек, но они также сложнее в обращении. Мы обсудим проблемы, связанные с одиночками, и поближе познакомимся с контроллерами StatefulSet в главе 11 «Служба с состоянием».

Как правило, приложения-одиночки, выполняющиеся в подах Kubernetes, открывают исходящие соединения с брокерами сообщений, реляционными базами данных, файловыми серверами или другими системами, работающими в других подах или внешних системах. Однако иногда под-одиночке может потребоваться принимать входящие соединения, и Kubernetes предлагает такую возможность в виде ресурса службы Service.

Подробно службы Service будут рассматриваться в главе 12 «Обнаружение служб», а здесь мы лишь кратко коснемся той их части, которая относится к одиночкам. Обычная служба Service (с параметром `type: ClusterIP`) создает виртуальный IP-адрес и балансирует нагрузку между всеми экземплярами, соответствующими селектору службы. Но под-одиночка, управляемый посредством StatefulSet, имеет единственный экземпляр и постоянную сетевую идентификацию. В таком случае лучше создать *автономную службу Service* (с параметрами `type: ClusterIP` и `clusterIP: None`). Она называется *автономной*, потому что не имеет виртуального IP-

адреса, kube-проxy не обслуживает такие службы и платформа не осуществляет их проксирование.

Однако такая служба все еще имеет практическую ценность, потому что автономная служба Service с селекторами создает конечные точки в API Server и генерирует записи A в DNS для соответствующих подов. Соответственно, поиск службы Service в DNS возвращает не ее виртуальный IP-адрес, а IP-адреса входящих в нее подов. Это обеспечивает прямой доступ к поду-одиночке через запись службы в DNS, минуя виртуальный IP-адрес службы. Например, если создать автономную службу Service с именем `my-singleton`, можно использовать ее как `my-singleton.default.svc.cluster.local` для прямого доступа к IP-адресу модуля.

Подводя итог, можно сказать, что когда требование к единственности экземпляра не является строгим, достаточно использовать контроллер ReplicaSet с одной репликой и обычную службу Service. Для строгого соблюдения требования и более эффективного обнаружения службы предпочтительнее использовать контроллер StatefulSet и автономную службу Service. Законченный пример этой конфигурации вы найдете в главе 11 «Служба с состоянием», где вам останется лишь уменьшить число реплик на одну, чтобы обеспечить выполнение единственного экземпляра.

Блокировка приложения изнутри

В распределенном окружении для управления количеством экземпляров службы часто используется распределенная блокировка, как показано на рис. 10.2. Всякий раз, когда активируется экземпляр службы или компонент в экземпляре, он может попытаться получить блокировку и в случае успеха перейти к активным действиям. Любой другой экземпляр службы, не сумевший получить блокировку, переходит в

состояние ожидания и постоянно повторяет попытки получить блокировку на случай, если текущая активная служба освободит ее.

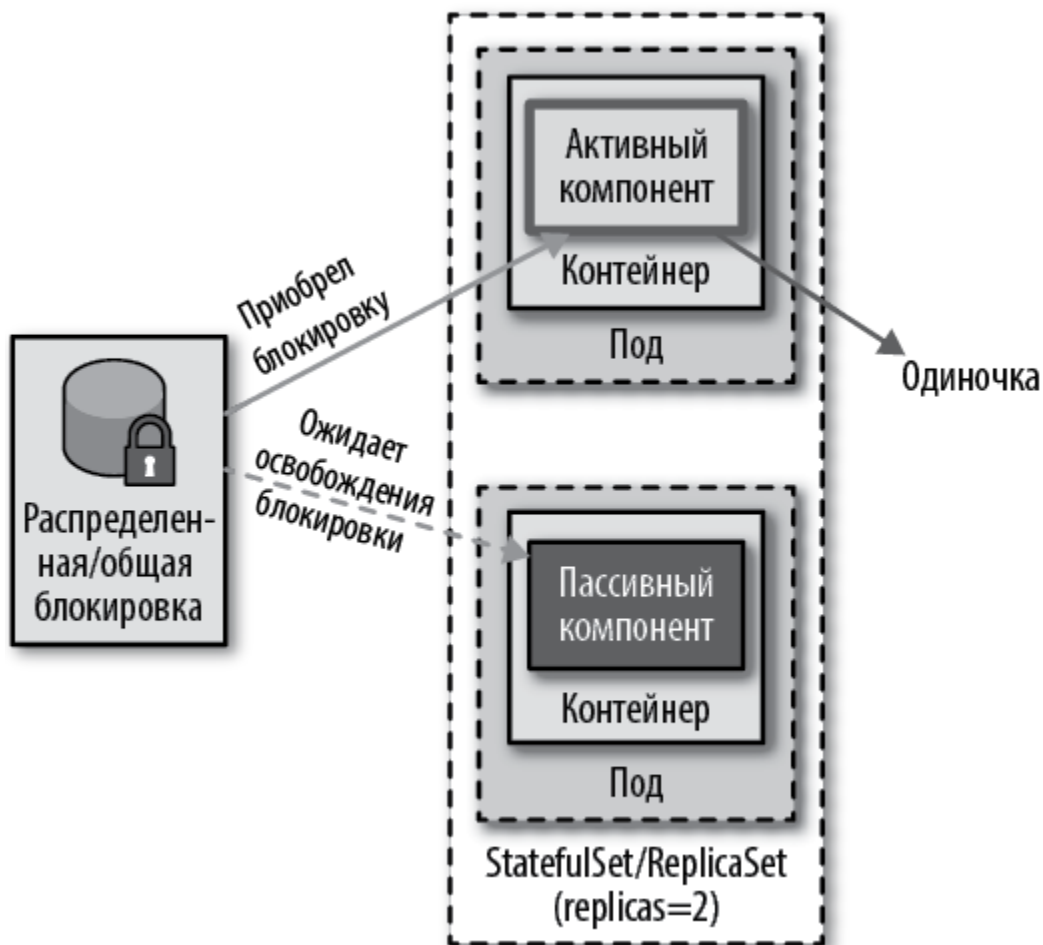


Рис. 10.2. Механизм блокировки приложения изнутри

Многие существующие распределенные платформы используют этот механизм для достижения высокой доступности и отказоустойчивости. Например, брокер сообщений Apache ActiveMQ может работать в высокодоступной топологии *активный-пассивный*, где общую блокировку обеспечивает источник данных. Экземпляр брокера, запустившийся первым, получает блокировку и становится активным, а другие экземпляры, запущенные

после, становятся пассивными и ожидают снятия блокировки. Эта стратегия обеспечивает наличие единственного активного экземпляра брокера, а также устойчивость к сбоям.

Эту стратегию можно сравнить с классической реализацией паттерна Singleton (Одиночка) в объектно-ориентированном мире: одиночка — это экземпляр объекта, хранящийся в статической переменной класса. В этом случае класс знает, что может существовать только один его экземпляр, и написан так, что не позволяет создавать несколько экземпляров в одном и том же процессе. В распределенных системах это означает, что само контейнерное приложение должно быть написано так, чтобы не было возможности иметь более одного активного экземпляра, независимо от количества запущенных экземпляров. Для этого в распределенном окружении должна иметься поддержка распределенных блокировок, например такая, как в Apache ZooKeeper, HashiCorp Consul, Redis или Etcd.

Типичная реализация в ZooKeeper использует эфемерные узлы, которые запускаются с началом сеанса клиента и удаляются по завершении сеанса. Первый запущенный экземпляр службы инициирует сеанс на сервере ZooKeeper и создает эфемерный узел, чтобы стать активным. Все остальные экземпляры в том же кластере становятся пассивными и должны ждать освобождения эфемерного узла. Именно так реализация на основе ZooKeeper обеспечивает наличие только одного активного экземпляра службы во всем кластере, обеспечивая поведение *активный-пассивный* для защиты от сбоев.

В мире Kubernetes вместо создания кластера ZooKeeper только ради поддержки блокировок лучше использовать механизм Etcd, доступный через Kubernetes API и выполняющийся на ведущих узлах. Etcd — это распределенное

хранилище пар ключ/значение, которое использует протокол Raft для поддержания своего реплицированного состояния. Особенно важно, что это хранилище предлагает необходимые строительные блоки для реализации выбора лидера, и уже имеется несколько клиентских библиотек, реализовавших эту возможность. Например, Apache Camel имеет компонент интеграции с Kubernetes, который реализует алгоритм выбора лидера и позволяет организовать выполнение единственного экземпляра. Кроме того, этот компонент вместо прямого доступа к Etcd API использует Kubernetes API и организует распределенную блокировку на основе карт конфигураций ConfigMap. Он полагается на гарантии оптимистичной блокировки в Kubernetes для правки таких ресурсов, как ConfigMap, позволяющей только одному поду обновлять конфигурацию в ConfigMap.

Реализация Camel использует эту гарантию, чтобы обеспечить активное выполнение только одного экземпляра маршрута Camel, а все остальные заставить ждать освобождения блокировки перед активацией. Это нестандартная реализация блокировки, но она позволяет достичь желаемого: при наличии нескольких подов с одним и тем же приложением Camel только один из них действует активно, а остальные ждут в пассивном режиме.

Реализация с использованием ZooKeeper, Etcd или любого другого механизма распределенной блокировки действует аналогично описанной: только один экземпляр приложения становится лидером и активизирует себя, а другие экземпляры находятся в пассивном ожидании освобождения блокировки. Это гарантирует, что даже если будет запущено несколько реплик пода и все выполняются исправно, только одна реплика будет активна и будет выполнять свои бизнес-функции, а

другие будут ждать приобретения блокировки, что может произойти в случае сбоя или выхода из строя ведущего узла.

Бюджет неработоспособности пода

В отличие от паттерна *Singleton Service* (Служба-одиночка) и механизма выбора лидера, которые пытаются ограничить максимальное количество экземпляров, действующих одновременно, ресурс бюджета неработоспособности пода `PodDisruptionBudget` в Kubernetes предлагает дополнительную и даже в чем-то противоположную возможность — возможность ограничить количество экземпляров, останавливаемых одновременно для обслуживания.

По своей сути `PodDisruptionBudget` гарантирует, что заданное количество или процент подов не будет добровольно вытеснено с узла. Под *добровольным* в данном случае понимается вытеснение, которое можно отложить на определенное время, — например, когда оно инициируется истощением ресурсов узла для обслуживания или обновления (`kubectl drain`) или уменьшением кластера, а не выходом узла из строя, что нельзя ни прогнозировать, ни контролировать.

В листинге 10.1 приводится определение ресурса `PodDisruptionBudget`, который применяется к подам, соответствующим селектору, и гарантирует доступность не менее двух подов в каждый момент времени.

Листинг 10.1. `PodDisruptionBudget`

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: random-generator-pdb
spec:
```

```
selector:  
  matchLabels: ❶  
    app: random-generator  
  minAvailable: 2 ❷
```

❶ Селектор для подсчета числа доступных подов.

❷ Не меньше двух подов должно быть доступно. В этом параметре также можно определить процент, например 80%, чтобы указать, что может быть вытеснено только 20% подов, соответствующих селектору.

Кроме `.spec.minAvailable` имеется также параметр `.spec.maxUnavailable`, определяющий максимальное количество подов, которые могут быть вытеснены. Но указать можно только какой-то один из параметров, а кроме того, `PodDisruptionBudget` обычно применяется только к подам, управляемым контроллером. Для подов, не управляемых контроллером (их также называют *простыми*, или *голыми*, подами), следует учитывать другие ограничения, связанные с `PodDisruptionBudget`.

Эта возможность может пригодиться в приложениях, работа которых основана на кворуме и для обеспечения кворума требуется одновременное выполнение некоторого минимального количества реплик. Или когда приложение обслуживает критический трафик и количество действующих экземпляров никогда не должно опускаться ниже определенного процента. Это еще один примитив Kubernetes, влияющий на управление экземплярами во время выполнения, и о нем стоило упомянуть в этой главе.

Пояснение

Когда требуются строгие гарантии единственности экземпляра, их нельзя получить, полагаясь на ReplicaSet и механизмы блокировки вне приложения. Назначение контроллера ReplicaSet в Kubernetes — гарантировать доступность подов, а не поддерживать семантику «не более одного». Как следствие, существует много сценариев нарушения требования единственности (например, когда узел, на котором запущен под-одиночка, отключается от остальной части кластера и отключенный экземпляр пода заменяется новым), когда в течение короткого периода времени одновременно действуют два пода. Если это неприемлемо, используйте контроллеры StatefulSet или рассмотрите варианты приобретения блокировок в приложении, которые предоставляют больший контроль над процессом выбора лидера с более строгими гарантиями. Подход на основе блокировок также поможет предотвратить случайное масштабирование подов при изменении значения параметра `replicas`.

Иногда требуется, чтобы только один компонент контейнерного приложения действовал в одиночку. Например, контейнерное приложение, реализующее конечную точку HTTP, можно безопасно масштабировать до нескольких экземпляров, но в нем имеется компонент опроса, который должен выполняться в одиночку. Использование подхода с блокировкой вне приложения не может использоваться в такой ситуации, так как помешает масштабированию всей службы. Мы должны либо выделить компонент-одиночку в отдельную единицу развертывания (что хорошо в теории, но не всегда практично из-за больших накладных расходов), либо использовать механизм блокировки внутри приложения и заблокировать только этот компонент. Это позволит прозрачно масштабировать все приложение и конечные точки HTTP, а для других частей использовать поведение *активный-пассивный*.

Дополнительная информация

- Пример службы-одиночки (<http://bit.ly/2TKp5nm>).
- Упрощенная реализация выборов лидера с использованием Kubernetes и Docker (<http://bit.ly/2FwUS1a>).
- Выборы лидера в клиенте на Go (<http://bit.ly/2UatejW>).
- Настройка бюджета неработоспособности пода (<http://bit.ly/2HDKcR3>).
- Создание служб-одиночек в кластере Kubernetes (<http://bit.ly/2TKm1HR>).
- Компонент интеграции Apache Camel с Kubernetes (<http://bit.ly/2JoL6mT>).

Глава 11. Служба с состоянием

Для распределенных приложений, имеющих хранимое состояние, требуются: неизменность идентификации, постоянные сетевые координаты, хранилище и упорядоченность. Паттерн *Stateful Service* (Служба с состоянием) описывает примитив `StatefulSet`, предлагающий все эти строительные блоки с надежными гарантиями, который идеально подходит для управления приложениями с состоянием.

Задача

К настоящему моменту мы познакомились со многими примитивами Kubernetes для создания распределенных приложений: контейнерами с поддержкой проверки работоспособности и ограничения ресурсов, группами контейнеров (подами), механизмом динамического размещения подов в пределах кластера, пакетными заданиями, инструментами планирования заданий, подами-одиночками и многими другими. Все эти примитивы характеризует одна общая черта: они рассматривают управляемое приложение как приложение без сохранения состояния, сконструированное из идентичных взаимозаменяемых контейнеров и соответствующее принципам методологии «Двенадцать факторов».

Наличие платформы, решающей задачи размещения, отказоустойчивости и масштабирования приложений без состояния, дает значительное преимущество, однако не менее важной является поддержка приложений с состоянием, каждый экземпляр которых уникален и имеет свои характеристики.

В реальном мире за каждой масштабируемой службой без состояния стоит служба с состоянием, обычно в форме некоторого хранилища данных. На ранних этапах развития Kubernetes, когда еще отсутствовала поддержка приложений с состоянием, эта проблема решалась путем размещения приложений без состояния в Kubernetes, а компонентов с состоянием — вне кластера, в общедоступном облаке или на локальном аппаратном обеспечении, управляемом с помощью традиционных, не облачных механизмов. Учитывая, что каждое предприятие имеет множество приложений с состоянием (устаревших и современных), отсутствие их поддержки считалось существенным ограничением Kubernetes, известной как универсальная облачная платформа.

Но какие типичные требования предъявляются приложениями с состоянием? Мы можем развернуть приложение с состоянием, такое как Apache ZooKeeper, MongoDB, Redis или MySQL, используя: развертывание Deployment, которое создает набор реплик ReplicaSet с параметром `replicas = 1`, чтобы обеспечить надежность приложения; службу Service для поддержки обнаружения его конечной точки; и PersistentVolumeClaim с PersistentVolume в роли хранилища для состояния.

В общем и целом это верно для приложения с состоянием, действующего в единственном экземпляре, но не совсем, потому что ReplicaSet не гарантирует семантику «не больше одного экземпляра», и количество реплик может увеличиваться на короткие промежутки времени. Это может иметь катастрофические последствия и приводить к потере данных. Кроме того, серьезные проблемы могут возникать в случае с распределенными службами с состоянием, состоящим из нескольких экземпляров. Приложение с состоянием, включающее несколько кластерных служб, требует от базовой

инфраструктуры разносторонних гарантий. Рассмотрим некоторые из наиболее распространенных требований, предъявляемых распределенными приложениями с сохранением состояния.

Хранилище

Мы можем увеличить количество реплик в `ReplicaSet` и получить распределенное приложение с состоянием. Но как определить требования к хранилищу в таком случае? Обычно для распределенного приложения с состоянием, такого как упомянутое выше, требуется выделенное постоянное хранилище для каждого экземпляра. Набор реплик `ReplicaSet` с параметром `replicas = 3` и определением `PersistentVolumeClaim (PVC)` приведет к тому, что все три пода будут подключены к одному и тому же постоянному тому `PersistentVolume (PV)`. `ReplicaSet` и `PVC` гарантируют лишь запуск требуемого числа экземпляров и подключение хранилища к узлам, где действуют эти экземпляры, но само хранилище не является выделенным, а совместно используется всеми экземплярами пода.

Решить проблему общего хранилища для всех экземпляров можно, реализовав механизм деления хранилища на сегменты и бесконфликтного их использования внутри самого приложения. Однако в этом случае создается единая точка отказа с единственным хранилищем. Кроме того, такой подход подвержен ошибкам из-за изменения количества подов в процессе масштабирования и может вызвать серьезные сложности в реализации предотвращения повреждения или потери данных во время масштабирования.

Другое решение — создание отдельного набора реплик `ReplicaSet` (с `replicas = 1`) для каждого экземпляра распределенного приложения с состоянием. В этом сценарии

каждый набор ReplicaSet получает свой запрос постоянного тома PVC и выделенное хранилище. Недостаток этого подхода — необходимость большого объема ручного труда: для масштабирования придется создать новый набор определений ReplicaSet, PVC или Service. В этом подходе не хватает всего одной абстракции, управляющей всеми экземплярами приложения с состоянием как единым целым.

Постоянные сетевые координаты

Кроме хранилища, распределенное приложение с состоянием требует постоянной идентификации в Сети. В дополнение к внутренним данным приложение с состоянием хранит такие сведения, как имя хоста и информация о соединениях со своими партнерами. То есть каждый экземпляр должен быть достижим по определенному адресу, который не должен динамически меняться, как IP-адреса подов в ReplicaSet. Это требование можно было бы удовлетворить с помощью обходного решения: создать Service и установить `replicas = 1` в ReplicaSet. Однако управление такой комбинацией должно осуществляться вручную, и само приложение не может полагаться на постоянство имени хоста, поскольку оно будет меняться после каждого перезапуска, а также ему будет неизвестно имя службы Service, открывающей доступ к нему.

Идентичность

Как следует из предыдущих требований, кластерные приложения с состоянием сильно зависят от наличия постоянных сетевых координат и выделенного долговременного хранилища для каждого экземпляра, потому что каждый экземпляр приложения с состоянием уникален и имеет свою идентичность, основными компонентами которой

являются долговременное хранилище и сетевые координаты. К этому списку также можно добавить идентификатор/имя экземпляра (для некоторых приложений с состоянием требуются уникальные постоянные имена), которое в Kubernetes будет именем пода. Поды, созданные с помощью ReplicaSet, получают произвольные имена и не сохраняют их после перезапуска.

Упорядоченность

Кроме уникальной и долговременной идентификации, экземпляры кластерных приложений с состоянием имеют фиксированное положение в их коллекциях. Обычно это положение влияет на последовательность масштабирования экземпляров вверх и вниз, но также может использоваться для распределения данных, блокировки или выбора ведущего экземпляра.

Другие требования

Постоянное и долговременное хранилище, постоянство сетевых координат и упорядоченность — все это типичные требования, предъявляемые кластерными приложениями с состоянием. Но управление приложениями с состоянием предъявляет также множество других специфических требований, которые меняются от случая к случаю. Например, некоторые приложения поддерживают понятие кворума и требуют постоянного присутствия некоторого минимального количества экземпляров; некоторые из них требуют последовательного развертывания экземпляров, тогда как другие поддерживают параллельное развертывание; некоторые допускают наличие дубликатов экземпляров, а некоторые нет. Предусмотреть все эти уникальные требования и предоставить

универсальный механизм просто невозможно, поэтому Kubernetes также позволяет создавать определения нестандартных ресурсов и операторов для управления приложениями с состоянием. Об операторах мы поговорим в главе 23.

Выше были перечислены некоторые типичные проблемы управления распределенными приложениями с состоянием и ряд не самых лучших вариантов их решения. А теперь давайте рассмотрим механизм, имеющийся в Kubernetes, помогающий удовлетворить эти требования, — примитив StatefulSet.

Решение

Чтобы объяснить, что предлагает StatefulSet для управления приложениями с состоянием, мы периодически будем сравнивать его с уже знакомым примитивом ReplicaSet, который в Kubernetes используется для выполнения приложений без состояния. Можно провести такую аналогию: примитив StatefulSet предназначен для управления домашними любимцами, а ReplicaSet — для управления домашним скотом. Домашние любимцы и скот — известная (хотя и неоднозначная) аналогия в мире DevOps: идентичные и взаимозаменяемые серверы называются скотом, а незаменимые уникальные серверы, требующие индивидуального ухода, называются домашними любимцами. Аналогично, примитив StatefulSet (первоначально основанный на этой аналогии и названный набором любимцев PetSet) предназначен для управления уникальными модулями, тогда как ReplicaSet предназначен для управления идентичными взаимозаменяемыми подами.

Давайте рассмотрим, как работают StatefulSet и как они удовлетворяют потребности приложений с состоянием. В

листинге 11.1 приводится определение нашей службы генератора случайных чисел в форме StatefulSet.[15](#)

Листинг 11.1. Служба Service для доступа к StatefulSet

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: rg ❶
spec:
  serviceName: random-generator ❷
  replicas: 2 ❸
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
        - image: k8spatterns/random-generator:1.0
          name: random-generator
          ports:
            - containerPort: 8080
              name: http
          volumeMounts:
            - name: logs
              mountPath: /logs
  volumeClaimTemplates: ❹
  - metadata:
      name: logs
```

```
spec:
  accessModes: [ "ReadWriteOnce" ]
  resources:
    requests:
      storage: 10Mi
```

❶ Имя набора StatefulSet используется как префикс при создании имен узлов.

❷ Ссылка на обязательную службу Service, которая определена в листинге 11.2.

❸ Два пода с именами *ng-0* и *ng-1*, являющихся членами набора StatefulSet.

❹ Паттерн создания PVC для каждого пода (подобно паттерну пода).

Мы не будем подробно разбирать листинг 11.1, а исследуем только общее его поведение и гарантии этого определения StatefulSet.

Хранилище

Большинство приложений с состоянием хранят некоторую информацию и поэтому требуют выделенного постоянного хранилища для каждого экземпляра. Настройка и связывание постоянного хранилища с подом в Kubernetes производится посредством PV и PVC. Для создания PVC вместе с подом StatefulSet использует элемент `volumeClaimTemplates`. Это дополнительное свойство является одним из основных отличий StatefulSet от примитива ReplicaSet, который имеет элемент `persistentVolumeClaim`.

Вместо ссылки на предопределенный PVC StatefulSet динамически создает PVC с помощью `volumeClaimTemplates` во время создания пода. Благодаря этому каждый под получает

свой выделенный PVC во время создания, а также во время масштабирования вверх, при изменении счетчика `replicas` в `StatefulSet`.

Как вы, наверное, заметили, мы говорили, что PVC создаются и связываются с подами, но мы ничего не сказали о PV. Дело в том, что примитив `StatefulSet` никак не управляет PV. Хранилище для подов должно быть заранее выделено администратором или провайдером PV на основе запрошенного класса хранения и готово для использования подами с состоянием.

Обратите внимание на асимметричное поведение: масштабирование `StatefulSet` вверх (увеличение счетчика `replicas`) создает новые поды и связанные с ними PVC. При масштабировании вниз поды удаляются, но PVC (и PV) никуда не исчезают, то есть постоянные тома PV не утилизируются и не удаляются, и Kubernetes не может освободить хранилище. Такое поведение основано на предположении, что хранилища для приложений с состоянием имеют важнейшее значение и периодическое масштабирование вниз не должно приводить к потере данных. Если вы уверены, что приложение с состоянием было остановлено специально и скопировало/передало свои данные другим экземплярам, то можете удалить PVC вручную, что позднее позволит повторно использовать соответствующий постоянный том PV.

Постоянные сетевые координаты

Каждый под, созданный контроллером `StatefulSet`, имеет постоянные и неизменные сетевые координаты, генерируемые на основе имени набора `StatefulSet` и порядкового индекса (начиная с 0). Так, предыдущий пример создаст два пода с именами `rg-0` и `rg-1`. Для генерации имен подов

используется четко установленный формат, который отличается от формата в ReplicaSet, основанного на добавлении случайного окончания.

Постоянные сетевые координаты являются такой же неотъемлемой чертой приложений с состоянием, как и наличие постоянного хранилища.

В листинге 11.2 определяется *автономная* (headless) служба Service. Параметр clusterIP: None здесь означает, что мы не хотим, чтобы эта служба обслуживалась маршрутизатором kube-проху, выделяла IP-адреса из диапазона адресов кластера или балансировала нагрузку. Возникает вопрос: зачем нужна эта служба?

Листинг 11.2. Служба Service для организации доступа к StatefulSet

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  clusterIP: None      ❶
  selector:
    app: random-generator
  ports:
    - name: http
      port: 8080
```

❶ Объявление службы автономной.

Поды без состояния, созданные с помощью ReplicaSet, идентичны, и поэтому для системы безразлично, кому из них передать запрос (балансировка нагрузки выполняется с помощью обычной службы Service). Но поды с состоянием

отличаются друг от друга, и иногда важно, чтобы запрос попал в определенный под с определенными координатами.

Автономная служба Service с селекторами (здесь `.selector.app == random-generator`) позволяет это сделать. Такая служба создает записи конечных точек Endpoint на API Server и записи A (адреса) в DNS, которые указывают непосредственно на поды, поддерживающие службу Service. Проще говоря, для каждого пода создается своя уникальная запись DNS, благодаря чему клиенты получают возможность напрямую связаться с ними. Например, если предположить, что наша служба `random-generator` принадлежит пространству имен `default`, мы можем связаться с подом `rg-0`, используя его полное доменное имя: `rg-0.random-generator.default.svc.cluster.local`, где имя пода добавляется в начало доменного имени службы. Такой формат позволяет другим компонентам кластерного приложения или другим клиентам напрямую обращаться к определенным подам.

Мы также можем выполнить поиск записей SRV в DNS (например, командой `dig SRV random-generator.default.svc.cluster.local`) и отыскать все работающие поды, зарегистрированные в управляющей службе Service, в наборе `StatefulSet`. Этот механизм позволяет любым приложениям динамически обнаруживать члены кластера. Связь между автономной службой Service и `StatefulSet` определяется не только селектором, но и ссылкой на имя службы в `StatefulSet`, в данном случае `serviceName: "random-generator"`.

Наличие выделенного хранилища, определяемого параметром `volumeClaimTemplates`, не является обязательным, но ссылка на службу в поле `serviceName` должна быть определена всегда. Управляющая служба Service

должна существовать до создания набора StatefulSet и отвечает за его сетевую идентификацию. При необходимости вы можете создать другие типы служб, которые дополнительно осуществляют балансировку нагрузки между подами с состоянием.

Как показано на рис. 11.1, наборы StatefulSet предлагает множество строительных блоков и гарантированный порядок управления приложениями с состоянием в распределенной среде, которые можно выбрать и использовать значимым для вас способом.

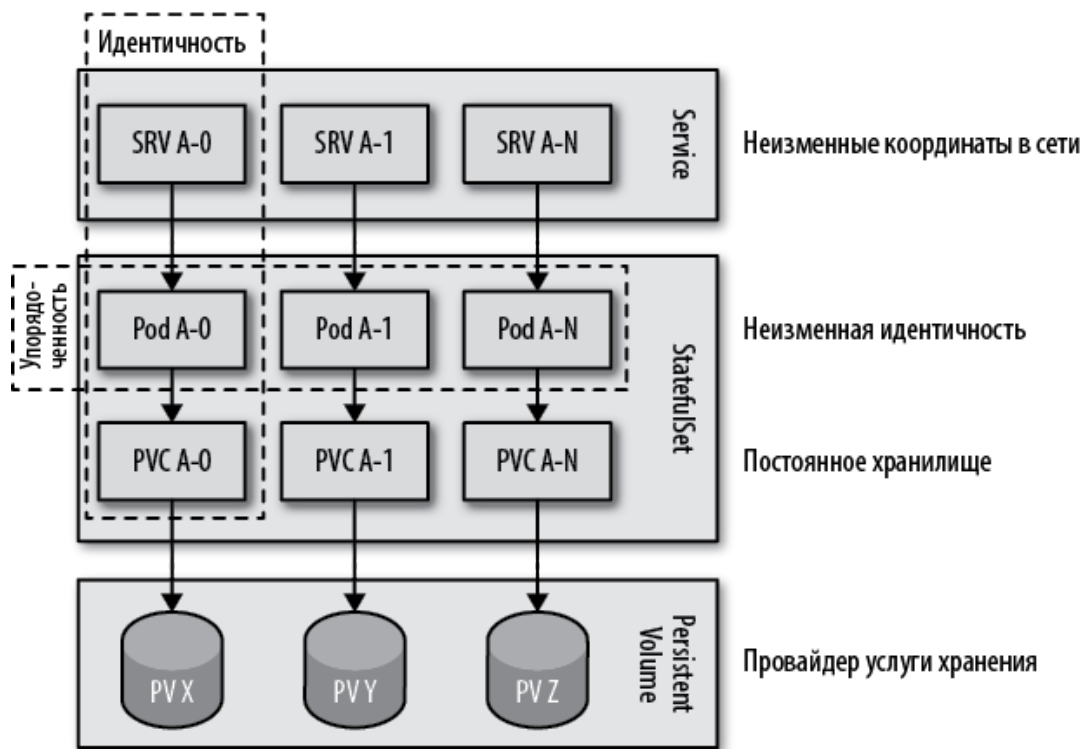


Рис. 11.1. Распределенное приложение с состоянием в Kubernetes

Идентичность

Идентичность — фундамент, на котором основываются все остальные гарантии контроллера StatefulSet. Опираясь на имя StatefulSet, можно получить имя пода и его идентичность.

Используя эту идентичность, можно определить имена PVC, обращаться к конкретными подам через автономные службы Service и т.д. Идентичность любого пода известна еще до его создания, и это знание можно использовать в самом приложении.

Упорядоченность

Распределенное приложение с состоянием по определению состоит из нескольких уникальных и невзаимозаменяемых экземпляров. Кроме их уникальности, экземпляры также могут быть связаны друг с другом порядком их создания, в соответствии с требованием *упорядоченности*.

С точки зрения StatefulSet упорядоченность важна только для масштабирования. Поды имеют имена, включающие порядковый индекс в конце (начиная с 0), и порядок их создания определяет порядок, в котором они масштабируются вверх и вниз (при масштабировании вниз поды останавливаются в обратном порядке следования их индексов, от $n - 1$ до 0).

Когда определяется ReplicaSet с несколькими репликами, поды планируются и запускаются все вместе, не дожидаясь успешного запуска предыдущих подов (под успешным запуском подразумевается переход в состояние готовности, как описывается в главе 4 «Проверка работоспособности»). Порядок запуска подов и их переход в состояние готовности не гарантируется. То же происходит, когда происходит масштабирование ReplicaSet вниз (изменением счетчика replicas или удалением набора). Все поды, принадлежащие ReplicaSet, начинают останавливаться одновременно без какого-либо порядка. Такое поведение обеспечивает быструю остановку, но может оказаться нежелательным для приложений с состоянием, особенно если имеют место

фрагментирование и распределение данных между экземплярами.

Чтобы обеспечить надлежащую синхронизацию данных в процессе масштабирования вверх или вниз, StatefulSet по умолчанию выполняет запуск и остановку подов последовательно. Это означает, что поды запускаются, начиная с первого (с индексом 0), и каждый последующий запускается только после успешного запуска предыдущего. Масштабирование вниз выполняется в обратном порядке: сначала останавливается под с самым большим индексом, и только после успешной остановки начинается остановка пода со следующим индексом в порядке убывания. Так продолжается, пока не остановится под с индексом 0.

Другие особенности

StatefulSet обладает еще целым рядом характеристик, которые можно настраивать в соответствии с потребностями приложений с состоянием. Каждое приложение с состоянием является уникальным и требует внимательного исследования перед попыткой вписать его в модель StatefulSet. Давайте рассмотрим еще несколько возможностей Kubernetes, которые могут пригодиться при работе с приложениями с состоянием.

Раздельное обновление

Мы описали выше гарантии упорядочения при масштабировании подов в StatefulSet. Для обновления запущенных приложений с состоянием (например, изменением параметра `.spec.template`), StatefulSet поддерживает поэтапное (такое, как канареечное) развертывание и гарантирует выполнение определенного

количества экземпляров, пока обновления будут применяться к остальной части экземпляров.

Организовать отдельное обновление экземпляров при использовании стратегии по умолчанию поэтапного обновления можно, указав число в параметре `.spec.updateStrategy.rollingUpdate.partition`.

Этот параметр (со значением по умолчанию 0) определяет порядковый номер, по которому следует разделить `StatefulSet` для обновления. Если параметр указан, обновятся все поды с индексами большими или равными значению `partition`, а остальные — нет. Это относится даже к остановленным подам; Kubernetes воссоздаст их в предыдущей версии. Это позволяет обновить сначала одну часть кластерного приложения с состоянием (например, чтобы гарантировать наличие кворума), а затем развернуть изменения в остальной части кластера, записав в параметр `partition` значение 0.

Параллельное развертывание

Когда параметру `.spec.podManagementPolicy` присваивается значение `Parallel`, контроллер `StatefulSet` запускает или завершает работу всех подов одновременно, не дожидаясь готовности или полной остановки предыдущего пода, прежде чем перейти к следующему. Если приложение не требует последовательной обработки, эта настройка может ускорить запуск или остановку.

Гарантированная семантика «не больше одного»

Уникальность — одно из фундаментальных свойств экземпляров приложений с состоянием, и Kubernetes

гарантирует, что никакие два пода в StatefulSet не будут иметь одинаковую идентичность или не будут привязаны к одному постоянному тому PV. Контроллер ReplicaSet, напротив, поддерживает семантику «гарантированного выполнения не менее X экземпляров одновременно». Например, ReplicaSet с двумя репликами стремится постоянно иметь не менее двух действующих экземпляров. Главная цель контроллера — не допустить уменьшения количества подов ниже заданного числа, даже ценой его увеличения в какие-то промежутки времени. При замене старого пода новым одновременно может выполняться большее число реплик, чем указано, когда новый под уже запустился, а старый еще не остановился полностью. Также число реплик может увеличиться, если узел Kubernetes будет классифицирован как недоступный с состоянием NotReady, но на нем все еще будут действовать ранее запущенные поды. В этом случае контроллер ReplicaSet запускает новые поды на исправных узлах, что может привести к увеличению числа работающих подов. Все это приемлемо в семантике *не менее X экземпляров*.

Контроллер StatefulSet, напротив, делает все возможное, чтобы гарантировать отсутствие дубликатов подов, то есть семантику «не более одного». Он не запустит под, пока не убедится, что старый экземпляр полностью остановлен. Когда узел выходит из строя, он планирует запуск новых подов на другом узле только после того, как Kubernetes подтвердит, что поды (и, возможно, весь узел) остановлены. Семантика *не более одного*, поддерживаемая контроллером StatefulSet, диктует эти правила.

Контроллер StatefulSet может нарушить эти гарантии и запустить дубликаты подов только при активном вмешательстве человека. Например, эту гарантию может

нарушить удаление объекта ресурса недоступного узла из API Server, пока физический узел продолжает работать. Такое действие должно выполняться, только если подтвердится, что узел не работает или выключен и на нем не запущены процессы пода. К нарушению гарантий может также привести принудительное удаление пода командой `kubectl delete pods _<pod>_ --grace-period = 0 --force`, которая не ждет подтверждения останова пода от Kubelet. Это действие немедленно удалит информацию о поде из API Server и заставит контроллер StatefulSet запустить заменяющий экземпляр пода, что может привести к дублированию.

Другие подходы для организации выполнения подов в единственном экземпляре обсуждаются в главе 10 «Служба-одиночка».

Пояснение

В этой главе мы рассмотрели некоторые стандартные требования и проблемы управления распределенными приложениями с состоянием. Мы выяснили, что организовать выполнение единственного экземпляра приложения с состоянием относительно просто, но обслуживание распределенного состояния — это сложная и многоплановая задача. Обычно мы связываем понятие «состояние» с понятием «хранилище», однако здесь мы увидели, что состояние может иметь несколько аспектов и разные приложения с состоянием могут требовать разных гарантий. В этом отношении контроллер StatefulSet является отличным средством для реализации распределенных приложений с состоянием. Он учитывает необходимость постоянного хранилища, работы в Сети (через сервисы), поддержки идентичности,

упорядоченности и некоторых других аспектов. Он предлагает хороший набор строительных блоков для автоматического управления приложениями с состоянием и превращает их в полноправных граждан облачного мира.

Контроллеры StatefulSet дают хорошее начало, но мир приложений с состоянием уникален и сложен. Кроме приложений с состоянием, разработанных для выполнения в облаке, которые прекрасно вписываются в StatefulSet, существует масса устаревших приложений, разработанных для выполнения в обычном, не облачном, окружении и предъявляющих еще более широкие требования. К счастью, фреймворку Kubernetes есть чем ответить на это. В сообществе Kubernetes давно поняли, что вместо моделирования различных рабочих нагрузок с помощью ресурсов Kubernetes и реализации их поведения с использованием универсальных контроллеров предпочтительнее позволить пользователям реализовать свои *контроллеры* и даже дать им возможность моделировать ресурсы приложений с помощью своих определений ресурсов и поведения с помощью *операторов*.

В главах 22 и 23 вы познакомитесь с соответствующими паттернами *Controller* (контроллер) и *Operator* (Оператор), которые с успехом можно использовать для управления сложными приложениями с состоянием в облачных окружениях.

Дополнительная информация

- Пример службы с состоянием (<http://bit.ly/2Y7SUN2>).
- Основы StatefulSet (<http://bit.ly/2r0boiA>).
- Документация с описанием StatefulSet (<http://bit.ly/2HGm6oE>).

- Развертывание Cassandra в StatefulSet (<http://bit.ly/2HBLNXA>).
- Запуск координатора распределенной системы ZooKeeper (<http://bit.ly/2JmNPNQ>).
- Автономные службы Service (<http://bit.ly/2v7Z19P>).
- Принудительное удаление подов в StatefulSet (<http://bit.ly/2OeuRrh>).
- Правильное масштабирование вниз приложений с состоянием в Kubernetes (<http://bit.ly/2Fk0mgK>).
- Настройка и развертывание приложений с состоянием (<http://bit.ly/2UsbkJt>).

[15](#) Предположим, что мы изобрели очень сложный способ получения случайных чисел в распределенном кластере RNG с несколькими экземплярами нашей службы в роли узлов. Конечно, это не так, но для примера это достаточно хорошая предпосылка.

Глава 12. Обнаружение служб

Паттерн *Service Discovery* (Обнаружение служб) предлагает постоянную конечную точку, с помощью которой клиенты могут получить доступ к экземпляру требуемой службы. Для этого Kubernetes поддерживает несколько механизмов, которые используются в разных конфигурациях, в зависимости от того, где располагаются службы и их потребители — в кластере или за его пределами.

Задача

Приложения, развернутые в Kubernetes, редко существуют сами по себе и обычно взаимодействуют с другими службами внутри кластера или с системами вне кластера. Взаимодействие может быть инициировано изнутри или извне. Взаимодействия, инициируемые изнутри, обычно выполняются методом опроса: приложение, сразу после запуска или позднее, подключается к другой системе и начинает посылать и получать данные. Типичным примером может служить приложение, запущенное в поде, которое устанавливает соединение с файловым сервером и начинает использовать файлы на нем, или подключается к брокеру сообщений и начинает получать либо отправлять сообщения, или соединяется с реляционной базой данных либо хранилищем пар ключ/значение и начинает читать или писать данные.

Важная отличительная особенность здесь заключается в том, что приложение, запущенное в поде, решает в какой-то момент открыть исходящее соединение с другим подом или внешней системой и начинает обмен данными. В этом

сценарии взаимодействия иницируются изнутри, и для приложения не нужны никакие дополнительные настройки в Kubernetes.

Этот подход часто используется для реализации паттернов, описанных в главе 7 «Пакетное задание» или в главе 8 «Периодическое задание». Кроме того, к другим системам иногда активно подключаются поды, действующие под управлением DaemonSet или ReplicaSet. Однако в Kubernetes наиболее распространен вариант, когда в кластере имеются службы, ожидающие соединений, чаще всего в форме входящих HTTP-соединений, от других подов в кластере или внешних систем. В этих случаях потребителям услуг необходим механизм для обнаружения подов, которые динамически размещаются планировщиком на узлах и иногда масштабируются вверх и вниз.

Это было бы серьезной проблемой, если бы нам пришлось самим отслеживать, регистрировать и искать конечные точки динамических подов в Kubernetes. Вот почему Kubernetes реализует паттерн *Service Discovery* (Обнаружение служб) с помощью различных механизмов, которые мы рассмотрим в этой главе.

Решение

В эпоху до появления Kubernetes наиболее распространенным механизмом обнаружения служб было обнаружение на стороне клиента. Когда потребителю требовалось обратиться к службе, которая может масштабироваться до нескольких экземпляров, потребитель должен был иметь агента обнаружения, способного отыскать в реестре экземпляры службы и выбрать один из них. Это решение может быть реализовано, например, с помощью встроенного агента (такого, как клиент ZooKeeper,

клиент Consul или Ribbon) или с помощью другого процесса, такого как Prana, предназначенного для поиска службы в реестре, как показано на рис. 12.1.

В эпоху после появления Kubernetes многие функции распределенных систем, такие как размещение, проверка и восстановление работоспособности, изоляция ресурсов, а также обнаружение служб и балансировка

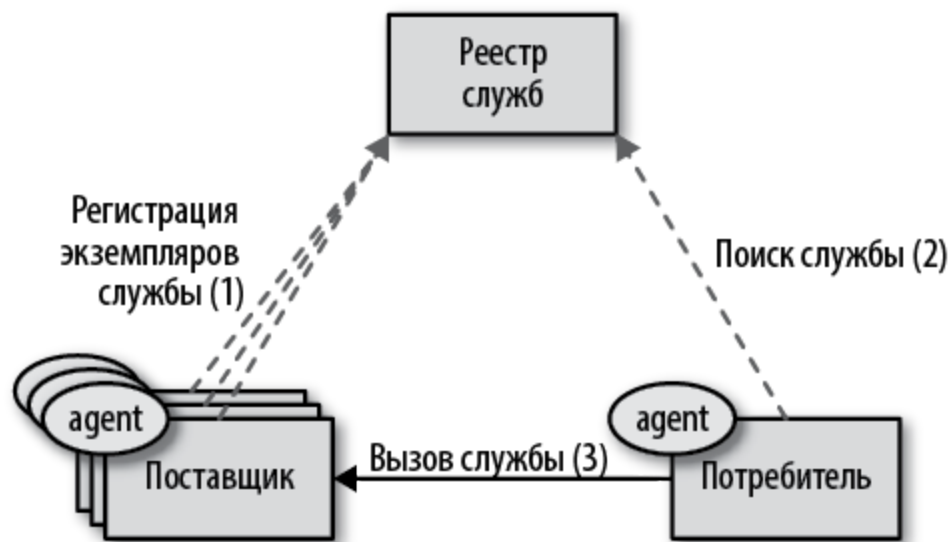


Рис. 12.1. Обнаружение служб на стороне клиента

нагрузки, были переложены на плечи платформы. Если использовать определения из сервис-ориентированной архитектуры (Service-Oriented Architecture, SOA), поставщик должен регистрировать каждый экземпляр в реестре, а потребитель должен обращаться к реестру, чтобы получить доступ к сервису.

В мире Kubernetes все это происходит за кулисами, поэтому потребитель службы вызывает фиксированную конечную точку виртуальной службы Service, которая динамически отыскивает экземпляры службы, реализованные в виде подов. На рис. 12.2 показано, как происходят регистрация и поиск в Kubernetes.

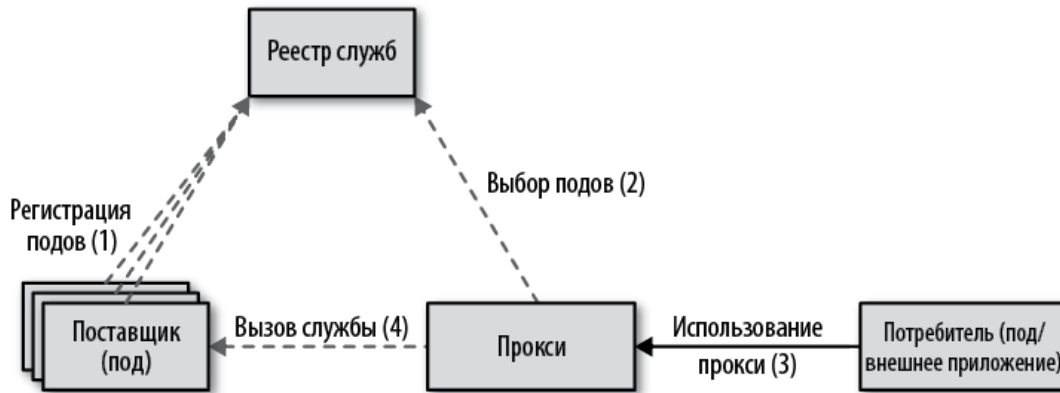


Рис. 12.2. Обнаружение служб на стороне сервера

На первый взгляд паттерн *Service Discovery* (Обнаружение служб) может показаться простым. Однако для его реализации можно использовать несколько механизмов в зависимости от того, где находится поставщик или потребитель — внутри кластера или за его пределами.

Обнаружение внутренних служб

Предположим, у нас есть веб-приложение и мы хотим запустить его в Kubernetes. Как только мы определим контроллер развертывания Deployment с несколькими репликами, планировщик разместит поды на подходящих узлах, и каждый из них получит IP-адрес кластера, назначенный до запуска. Если другая клиентская служба в другом модуле пожелает обратиться к конечным точкам веб-приложения, у нее не будет простого способа заранее узнать IP-адреса подов с приложением.

В Kubernetes эту проблему решает ресурс Service. Он организует постоянную точку входа для коллекции подов с одинаковой функциональностью. Самый простой способ создать Service — выполнить команду `kubectl expose`, которая создаст Service для пода или нескольких подов в Deployment или ReplicaSet. Команда создаст виртуальный IP-

адрес, который запишет в параметр `clusterIP`, и извлечет из ресурсов селекторы подов и номера портов, чтобы создать определение службы `Service`. Однако чтобы получить более полный контроль, лучше создать определение `Service` вручную, как показано в листинге 12.1.

Листинг 12.1. Простая служба `Service`

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  selector: ❶
    app: random-generator
  ports:
    - port: 80 ❷
      targetPort: 8080 ❸
      protocol: TCP
```

❶ Селектор, описывающий метки соответствующих ему подов.

❷ Порт, через который доступна эта служба `Service`.

❸ Порт, который прослушивают поды.

Этот пример определяет службу `Service` с именем `random-generator` (имя потребуется позже для обнаружения) и типом `type: ClusterIP` (по умолчанию), которая принимает TCP-соединения через порт 80 и направляет их в порт 8080 всех соответствующих подов с селектором `app: random-generator`. Неважно, когда и как создаются поды, — любой под, соответствующий селектору, становится целью маршрутизации, как показано на рис. 12.3.

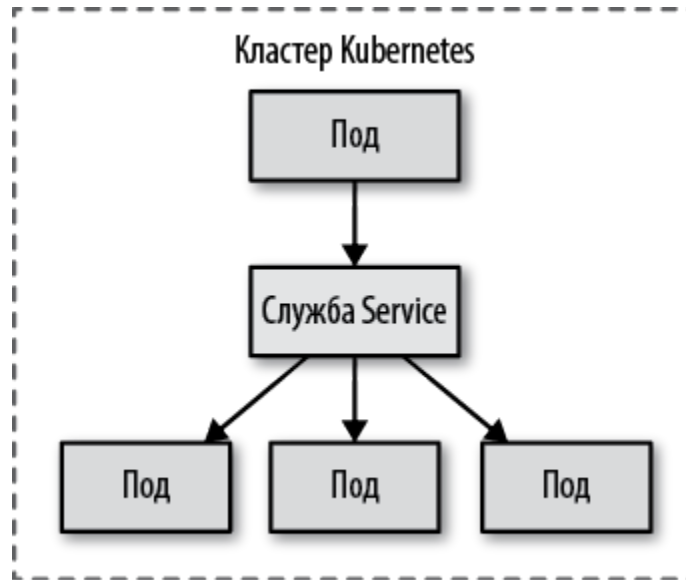


Рис. 12.3. Обнаружение внутренней службы

Важно помнить, что после создания службы Service она получит `clusterIP`, доступный только внутри кластера Kubernetes (на что указывает имя параметра), и этот IP-адрес остается неизменным, пока существует определение службы. Но как другие приложения в кластере смогут определить этот динамически распределенный `clusterIP`? В их распоряжении есть два способа:

Через переменные окружения

Запуская поды, Kubernetes заполняет их переменные окружения сведениями обо всех службах Service, действовавших до этого момента. Например, наша служба `random-generator`, прослушивающая порт 80, будет внедряться во все вновь запускаемые поды через переменные окружения, как показано в листинге 12.2. Приложение, выполняющееся в этом поде, сможет узнать, какое имя службы использовать, обратившись к своим переменным окружения. Этот простой механизм поиска

можно использовать в приложениях, написанных на любом языке, а кроме того, его легко симитировать за пределами кластера Kubernetes для целей разработки и тестирования. Основной проблемой этого механизма является временная зависимость от создания службы Service. Поскольку переменные среды нельзя добавить в уже запущенные поды, координаты службы будут доступны только подам, запущенным после создания службы в Kubernetes. Соответственно, служба Service должна быть определена до запуска подов, зависящих от нее, или, если это невозможно, поды необходимо перезапустить.

Листинг 12.2. Переменные окружения с информацией о службах, автоматически создаваемые в подах

```
RANDOM_GENERATOR_SERVICE_HOST=10.109.72.32  
RANDOM_GENERATOR_SERVICE_PORT=8080
```

Через поиск в DNS

В Kubernetes действует свой DNS-сервер, и все поды автоматически настраиваются на его использование. Более того, когда создается новая служба Service, для нее автоматически создается новая запись в DNS. Если предположить, что клиент знает имя необходимой ему службы, он сможет обратиться к ней по полному доменному имени (Fully Qualified Domain Name, FQDN), такому как `random-generator.default.svc.cluster.local`. Здесь `random-generator` — это имя службы Service, `default` — имя пространства имен, `svc` указывает, что это ресурс Service, а `cluster.local` — окончание, характерное для кластера. При обращении к службе в том же пространстве имен окончание с именем кластера, а также пространство имен можно опустить.

Механизм обнаружения с использованием DNS не имеет недостатков, свойственных переменным окружения, потому что сервер DNS позволяет любым подам отыскивать все службы Service сразу после их определения. Однако вам все равно может понадобиться использовать переменные окружения для определения номера порта, если он нестандартный или неизвестен потребителю службы.

Вот еще несколько основных характеристик служб Service с типом `type: ClusterIP`, на которых основываются другие типы:

Несколько портов

В одном определении службы можно зарезервировать несколько портов, исходных и целевых. Например, если под поддерживает возможность соединения по протоколу HTTP через порт 8080 и по протоколу HTTPS через порт 8443, нет необходимости определять две службы, потому что одна служба Service может связать оба порта с портами 80 и 443.

Близость сеансов

Когда появляется новый запрос, Service выбирает случайный под. Это поведение можно изменить, определив параметр `sessionAffinity: ClientIP`, который привяжет все запросы, исходящие с одного и того же IP-адреса, к определенному поду. Не забывайте, что службы Service в Kubernetes выполняют балансировку нагрузки на транспортном уровне L4 и не могут исследовать сетевые пакеты и выполнять балансировку нагрузки на уровне приложений, например на основе cookie.

Проверка готовности

В главе 4 «Проверка работоспособности» вы узнали, как определить параметр `readinessProbe` для контейнера. Если для пода определены проверки готовности и они терпят неудачу, под удаляется из списка конечных точек службы Service, даже если он соответствует селектору.

Виртуальный IP-адрес

При создании служба Service с типом `type: ClusterIP` получает постоянный виртуальный IP-адрес. Однако этот IP-адрес не соответствует ни одному сетевому интерфейсу и не существует в реальности. Он обслуживается компонентом `kube-proxy`, действующим на каждом узле, который получает информацию о новой службе и обновляет настройки `iptables` на своем узле, добавляя правила, которые перехватывают сетевые пакеты, направляющиеся на этот виртуальный IP-адрес, и заменяют его IP-адресом выбранного пода. В настройки `iptables` не добавляются правила для ICMP — только для протокола, указанного в определении службы, такого как TCP или UDP. Как следствие, нет возможности использовать `ping` для проверки IP-адреса службы, потому что эта команда использует протокол ICMP. Зато есть возможность получить доступ к службе через TCP (например, послать HTTP-запрос).

Выбор ClusterIP

В определении службы Service можно явно указать IP-адрес в поле `.spec.clusterIP`. Это должен быть действительный IP-адрес из predeterminedного диапазона. Хотя это и не рекомендуется, но этот параметр может пригодиться при

работе с устаревшими приложениями, настроенными на использование определенного IP-адреса, или при наличии существующей записи DNS, которую желательно использовать повторно.

Службы Service с типом `type: ClusterIP` доступны только внутри кластера Kubernetes, используются для обнаружения подов, соответствующих селекторам, и применяются наиболее часто. Далее мы рассмотрим другие типы служб, которые позволяют обнаруживать конечные точки, указанные вручную.

Ручное обнаружение служб

Когда создается служба Service с параметром `selector`, Kubernetes создает и поддерживает список подходящих и готовых к использованию подов в списке ресурсов конечных точек. Выполнив команду `kubectl get endpoints random-generator`, можно убедиться, что были созданы все конечные точки от имени службы в листинге 12.1. Подключения можно перенаправлять не только к подам внутри кластера, но и к внешним IP-адресам и портам. Для этого следует опустить определение `selector` в Service и вручную создать ресурсы конечных точек, как показано в листинге 12.3.

Листинг 12.3. Service без параметра `selector`

```
apiVersion: v1
kind: Service
metadata:
  name: external-service
spec:
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 80
```

Далее, в листинге 12.4 определяется ресурс конечных точек с именем, совпадающим с именем службы Service, который определяет целевые IP-адреса и порты.

Листинг 12.4. Конечные точки для внешней службы

```
apiVersion: v1
kind: Endpoints
metadata:
  name: external-service ❶
subsets:
  - addresses:
    - ip: 1.1.1.1
    - ip: 2.2.2.2
  ports:
    - port: 8080
```

❶ Имя должно совпадать с именем службы Service, обеспечивающей доступ к этим конечным точкам Endpoints.

Кроме того, эта служба Service доступна только изнутри кластера и может использоваться так же, как предыдущие, через переменные окружения или поиск в DNS. Разница лишь в том, что список конечных точек поддерживается вручную и их IP-адреса обычно находятся вне кластера, как показано на рис. 12.4.

Это не единственный механизм подключения к внешним ресурсам, хотя и используется чаще других. Конечные точки могут содержать IP-адреса подов, но не виртуальные IP-адреса других служб Service. Одной из



Рис. 12.4. Обнаружение службы вручную

замечательных особенностей служб Service является возможность добавлять и удалять селекторы и ссылаться на внешних или внутренних провайдеров без удаления определения ресурса, что может привести к изменению IP-адреса службы. То есть потребители услуг могут продолжать использовать тот же IP-адрес службы, даже если фактическая реализация поставщика услуг будет перемещаться по локальной сети Kubernetes.

В категории методов ручного обнаружения служб есть еще тип ресурса Service, представленный в листинге 12.5.

Листинг 12.5. Служба Service с внешним адресом назначения

```
apiVersion: v1
kind: Service
metadata:
  name: database-service
spec:
  type: ExternalName
  externalName: my.database.example.com
ports:
```

- port: 80

Эта служба Service тоже не имеет параметра selector, но ее тип определен как ExternalName. Это важное отличие с точки зрения реализации. Эта служба ссылается на ресурс с именем, указанным в externalName, адрес которого определяется исключительно с помощью DNS. Это — способ создания псевдонима для внешней конечной точки с использованием DNS CNAME вместо прокси с IP-адресом. Но по сути это еще один способ абстрагирования конечных точек, расположенных за пределами кластера Kubernetes.

Обнаружение служб в кластере извне

Все механизмы обнаружения служб, представленные выше в этой главе, используют виртуальный IP-адрес, указывающий на поды или внешние конечные точки, а сам виртуальный IP-адрес доступен только внутри кластера Kubernetes. Однако всякий кластер Kubernetes работает в реальном мире, и кроме подключения его подов к внешним ресурсам часто требуется обратное — возможность подключения внешних приложений к конечным точкам подов. Давайте посмотрим, как сделать под доступным для клиентов, действующих за пределами кластера.

Первый способ — определить службу Service и экспортировать ее за границы кластера, задав тип type: NodePort.

Определение в листинге 12.6 описывает службу Service, поддерживаемую подами, которые соответствуют селектору app: random-generator. Она принимает соединения через порт 80 по виртуальному IP-адресу и направляет их на порт 8080 выбранного пода. Но дополнительно это определение резервирует порт 30036 на всех узлах и перенаправляет все входящие соединения через этот порт в службу Service. Это

резервирование обеспечивает доступность службы и через виртуальный IP-адрес, и извне, через выделенный порт на каждом узле.

Листинг 12.6. Служба Service с типом NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  type: NodePort ❶
  selector:
    app: random-generator
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30036 ❷
      protocol: TCP
```

❶ Открывает порт на всех узлах.

❷ Фиксированный номер порта (который должен быть доступен). Если опустить этот параметр, будет выбран случайный номер порта.

Этот метод экспортирования служб (как показано на рис. 12.5) может показаться неплохим вариантом, но у него есть свои недостатки. Давайте рассмотрим некоторые из его отличительных характеристик:

Номер порта

Вместо выбора конкретного порта в виде параметра `nodePort: 30036` можно позволить Kubernetes

самостоятельно выбрать порт из его диапазона.

Правила брандмауэра

Так как этот метод основан на открытии порта на всех узлах, в брандмауэр необходимо добавить дополнительные правила, разрешающие внешним клиентам подключаться к этому порту.

Выбор узла

Внешний клиент может открыть соединение с любым узлом в кластере. Но если узел окажется недоступным, клиентское приложение должно само установить соединение с другим работоспособным узлом. С этой целью может быть полезно установить балансировщик нагрузки перед узлами, который выбирает исправные узлы и обрабатывает отказы.

Выбор пода

Когда клиент открывает соединение через порт узла, он перенаправляется к случайно выбранному поду, который может находиться на том же узле, где было открыто соединение, или на другом узле.

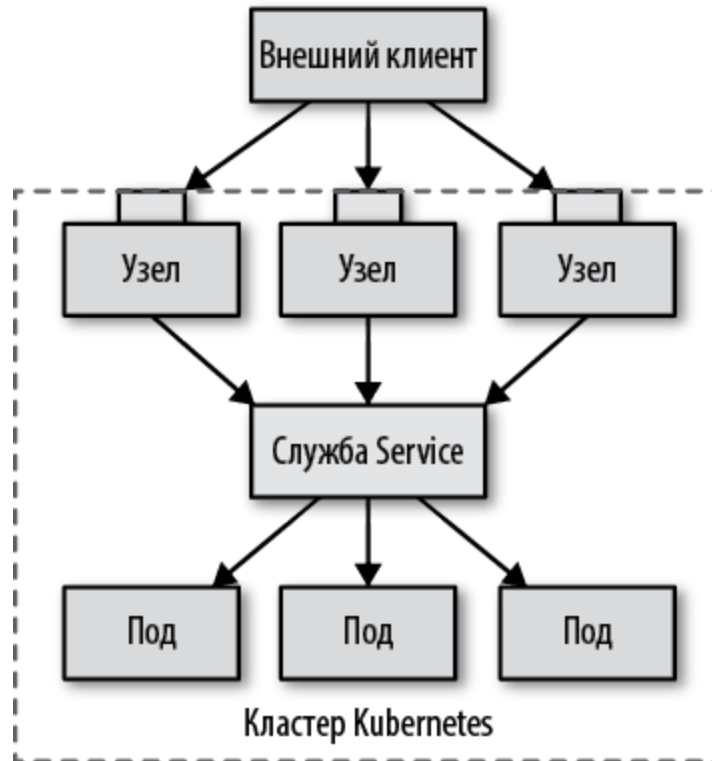


Рис. 12.5. Обнаружение служб по порту узла

Этой дополнительной переадресации можно избежать и заставить Kubernetes всегда выбирать под на узле, где было открыто соединение, если добавить параметр `externalTrafficPolicy: Local` в определение Service. В этом случае Kubernetes не позволит подключаться к модулям, размещенным на других узлах, что, впрочем, может породить проблемы. Чтобы решить их, нужно гарантировать размещение подов на всех узлах (например, реализовав паттерн *Daemon Service* (Фоновая служба)) или предоставить клиенту информацию о том, на каких узлах имеются исправные поды.

Исходящие адреса

Существуют еще некоторые особенности, связанные с исходящими адресами пакетов, отправляемых службам

Service разных типов. В частности, службы типа NodePort получают пакеты с исходящими адресами клиентов, прошедшими процедуру преобразования сетевых адресов (NAT), то есть исходящие IP-адреса клиентов в сетевых пакетах заменяются внутренними адресами узла. Например, когда клиентское приложение отправляет пакет на узел 1, исходящий адрес в нем меняется на адрес узла, адрес назначения меняется на адрес пода, после чего пакет пересылается на узел 2, где находится сам под. Когда под получает сетевой пакет, исходящий адрес в нем будет отличаться от адреса оригинального клиента и совпадать с адресом узла 1. Чтобы этого не происходило, можно определить параметр `externalTrafficPolicy: Local`, как описано выше, и передавать трафик только подам, находящимся на узле 1.

Другой способ организовать возможность обнаружения служб для внешних клиентов — настроить балансировку нагрузки. Вы уже видели, как служба Service с типом `type: NodePort` строится поверх обычной службы с типом `type: ClusterIP`, дополнительно открывая порт на каждом узле. Ограничением этого подхода является необходимость иметь балансировщик нагрузки для клиентских приложений, выбирающий работоспособный узел. Службы с типом `LoadBalancer` устраняют это ограничение.

Кроме создания обычных служб Service и открытия портов на каждом узле с помощью служб с типом `type: NodePort`, можно также экспортировать службу вовне с использованием балансировщика нагрузки облачного провайдера. На рис. 12.6 показана такая конфигурация: собственный балансировщик нагрузки играет роль шлюза в кластер Kubernetes.

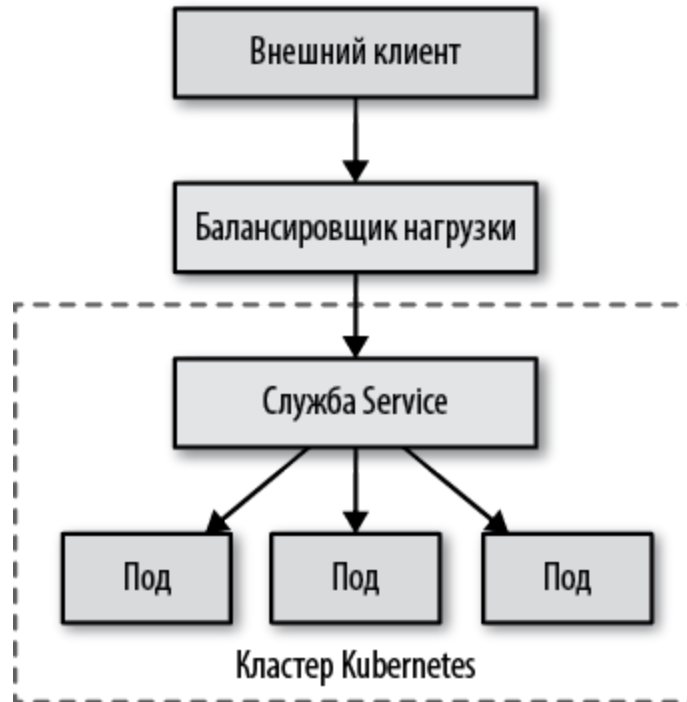


Рис. 12.6. Обнаружение служб посредством балансировщика нагрузки

То есть службы Service этого типа действуют, только когда облачный провайдер настроил в Kubernetes поддержку балансировки нагрузки.

Мы можем создать службу с балансировщиком нагрузки, указав тип `LoadBalancer`. Встретив такую службу, Kubernetes добавит IP-адреса в поля `.spec` и `.status`, как показано в листинге 12.7.

Листинг 12.7. Служба Service типа `LoadBalancer`

```

apiVersion: v1
kind: Service
metadata:
  name: random-generator
spec:
  type: LoadBalancer
  clusterIP: 10.0.171.239
  loadBalancerIP: 78.11.24.19
  
```

```
selector:
  app: random-generator
ports:
- port: 80
  targetPort: 8080
  protocol: TCP
status: ②
loadBalancer:
  ingress:
  - ip: 146.148.47.155
```

① Kubernetes присваивает clusterIP и loadBalancerIP, если они доступны.

② Поле status управляется фреймворком Kubernetes и добавляет Ingress IP.

С этим определением службы внешнее клиентское приложение может открыть соединение с балансировщиком нагрузки, который сам выберет узел и найдет под. Способ предоставления доступа к балансировщику нагрузки и обнаружения служб зависит от поставщика облачных услуг. Некоторые облачные провайдеры разрешают определять адрес балансировщика нагрузки, а некоторые — нет. Некоторые поддерживают механизмы сохранения исходящих адресов, а некоторые заменяют их адресом балансировщика. Обязательно проверяйте конкретную реализацию, предлагаемую выбранным вами облачным провайдером.



Поддерживается еще один тип служб Service: автономные (headless) службы, для которых не требуется

выделенный IP-адрес. Автономная служба создается добавлением параметра `clusterIP: None` в раздел `spec:`. Поддерживаемые поды добавляются в автономную службу внутренним DNS-сервером, и такие службы наиболее полезны для реализации `StatefulSet`, как подробно описано в главе 11 «Служба с состоянием».

Обнаружение служб уровня приложения

В отличие от механизмов, обсуждавшихся до сих пор, `Ingress` — это не тип службы, а самостоятельный ресурс `Kubernetes`, который располагается перед службами `Service` и действует подобно интеллектуальному маршрутизатору и точке входа в кластер¹⁶. Обычно ресурс `Ingress` используется с целью открыть доступ к службам `Service` по протоколу `HTTP` через внешние `URL` и обеспечить балансировку нагрузки, завершение `SSL` (`SSL termination`) и виртуальный хостинг на основе имен, но существуют и другие специализированные реализации `Ingress`.

Для нормальной работы `Ingress` необходимо, чтобы в кластере был запущен один или несколько контроллеров `Ingress`. В листинге 12.8 показан простой ресурс `Ingress`, экспортирующий единственную службу `Service`.

Листинг 12.8. Определение простого ресурса `Ingress`

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: random-generator
spec:
  backend:
    serviceName: random-generator
```

```
servicePort: 8080
```

В зависимости от инфраструктуры, на которой работает Kubernetes, и реализации контроллера Ingress, это определение выделяет IP-адрес, доступный извне, и открывает доступ к службе `random-generator` через порт 80. Но эта конфигурация не сильно отличается от службы Service с типом `type: LoadBalancer`, которая требует внешнего IP-адреса для каждого определения Service. Настоящая сила Ingress заключается в использовании единого внешнего балансировщика нагрузки и IP-адреса для обслуживания нескольких служб и снижения затрат на инфраструктуру.

В листинге 12.9 показана простая конфигурация веерной маршрутизации между несколькими службами с единым IP-адресом, основанная на HTTP URI.

Листинг 12.9. Определение ресурса Ingress с отображением маршрутов

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: random-generator
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target:
/
spec:
  rules:
    - http:
      paths:
        - path: /
          backend:
            serviceName: random-generator
            servicePort: 8080
        - path: /cluster-status
```

```
backend:  
  serviceName: cluster-status  
  servicePort: 80
```

- ❶ Специализированные правила маршрутизации запросов на основе пути для контроллера Ingress.
- ❷ Отправлять в службу random-generator все запросы...
- ❸ ...кроме запросов, в которых указан путь /cluster-status; эти запросы должны передаваться другой службе.

маршруты в OpenShift

Red Hat OpenShift – популярный корпоративный дистрибутив Kubernetes. Помимо полной совместимости с Kubernetes, OpenShift предлагает ряд дополнительных возможностей. Одна из них – маршруты Route, очень похожие на Ingress. Они настолько похожи, что почти неразличимы. Следует отметить, что появление поддержки маршрутов Route предшествовало введению объекта Ingress в Kubernetes, поэтому маршруты Route можно смело считать прямыми предшественниками Ingress.

Тем не менее между объектами Route and Ingress существуют технические отличия:

Маршрут Route автоматически поддерживается балансировщиком нагрузки HAProxy, интегрированным в OpenShift, поэтому нет необходимости устанавливать дополнительный контроллер Ingress. Однако в OpenShift есть возможность заменить встроенный балансировщик нагрузки.

Имеется возможность использовать дополнительные режимы завершения TLS, такие как повторное шифрование и сквозной доступ к службе.

Поддерживается распределение трафика между экземплярами с учетом весовых коэффициентов.

Поддерживаются шаблонные символы в доменных именах.

Наконец, Ingress тоже можно использовать в OpenShift. То есть, используя OpenShift, вы получаете некоторую свободу выбора.

Поскольку все контроллеры Ingress имеют разные реализации, кроме обычного определения ресурса Ingress контроллеру может потребоваться дополнительная информация, которая передается посредством аннотаций. Если предположить, что контроллер Ingress настроен правильно, предыдущее определение обеспечит использование балансировки нагрузки и доступность двух служб с разными путями через один IP-адрес, как показано на рис. 12.7.



Рис. 12.7. Обнаружение служб уровня приложения

Ingress — это самый мощный и в то же время самый сложный механизм обнаружения служб в Kubernetes. Главное его преимущество — возможность организовать доступ к нескольким службам через один IP-адрес, когда все службы используют один и тот же протокол прикладного уровня L7 (обычно HTTP).

Пояснение

В этой главе мы рассмотрели основные механизмы, используемые при реализации паттерна *Service Discovery* (Обнаружение служб) в Kubernetes. Обнаружение динамических подов внутри кластера всегда производится через ресурс Service, хотя разные варианты могут приводить к разным реализациям. Абстракция Service — это высокоуровневый облачный способ настройки низкоуровневых деталей, таких

как виртуальные IP-адреса, правила iptables, записи DNS или переменные окружения. Обнаружение служб извне кластера основывается на абстракции Service и направлено на предоставление услуг внешнему миру. Тип NodePort позволяет определить простую службу, доступную извне, однако для настройки служб с высокой доступностью требуется интеграция с поставщиком инфраструктуры платформы.

В табл. 12.1 перечислены разные способы реализации паттерна *Service Discovery* (Обнаружение служб) в Kubernetes. В ней дается краткая информация о различных механизмах обнаружения служб, описанных в этой главе, организованных в порядке возрастания сложности. Мы надеемся, что она поможет вам мысленно построить модель и лучше понять ее.

Таблица 12.1. Механизмы паттерна Service Discovery (Обнаружение служб)

Название	Конфигурация	Тип клиента	Описание
ClusterIP	type: ClusterIP .spec.selector	Внутренний	Наиболее распространенный механизм обнаружения служб изнутри
Ручное обнаружение служб по IP	type: ClusterIP kind: Endpoints	Внутренний	Обнаружение по внешнему IP-адресу
Обнаружение по полному доменному имени	type: ExternalName .spec.externalName	Внутренний	Обнаружение по внешнему полному доменному имени
Автономная служба	type: ClusterIP .spec.clusterIP: None	Внутренний	Обнаружение на основе DNS без виртуального IP-адреса
NodePort	type: NodePort	Внешний	Предпочтительный вариант для трафика, отличного от HTTP
LoadBalancer	type: LoadBalancer	Внешний	Требует поддержки со стороны облачной

			инфраструктуры
Ingress	kind: Ingress	Внешний	Интеллектуальный механизм маршрутизации на основе L7/HTTP

В этой главе представлен достаточно полный обзор всех основных механизмов Kubernetes для обнаружения служб и доступа к ним. Однако наше путешествие на этом не заканчивается. В рамках проекта *Knative* были созданы новые примитивы поверх Kubernetes, которые помогают разработчикам приложений организовать обслуживание и обмен сообщениями.

В контексте паттерна *Service Discovery* (Обнаружение служб) проект *Knative Serving* представляет особый интерес, потому что предлагает новый ресурс, напоминающий Service, который был представлен здесь, но с другой группой API. Knative Serving предлагает также поддержку ревизий приложений и очень гибкое масштабирование служб за балансировщиком нагрузки. Мы кратко расскажем о Knative Serving в разделе «Knative Build» главы 25 и во врезке «Knative Serving» главы 24, однако полное обсуждение Knative выходит за рамки этой книги. В разделе «Дополнительная информация» главы 24 вы найдете ссылки, где можно найти более подробную информацию о Knative.

Дополнительная информация

- Пример обнаружения служб (<http://bit.ly/2TeXzcr>).
- Службы Service в Kubernetes (<http://bit.ly/2q7AbUD>).
- Использование DNS для обнаружения служб и подов (<http://bit.ly/2Y5jUwL>).
- Отладка служб Service (<http://bit.ly/2r0igMX>).

- Использование исходящего IP-адреса (<https://kubernetes.io/docs/tutorials/services/>).
- Настройка внешнего балансировщика нагрузки (<http://bit.ly/2Gs05Wh>).
- Выбор между NodePort, LoadBalancer и Ingress в Kubernetes (<http://bit.ly/2GrVio2>).
- Ingress (<https://kubernetes.io/docs/concepts/services-networking/ingress/>).
- Выбор между Kubernetes Ingress и OpenShift Route (<https://red.ht/2JDDflo>).

¹⁶ Название ресурса Ingress так и переводится — *вход, точка входа*. — *Примеч. пер.*

Глава 13. Самоанализ

Некоторые приложения должны иметь возможность получать информацию о себе. Паттерн *Self Awareness* (Самоанализ) описывает Kubernetes Downward API — простой механизм для самоанализа и внедрения метаданных в приложения.

Задача

Большинство облачных приложений не имеют состояния и отличительной идентичности. Однако иногда даже этим приложениям бывает необходима информация о себе и об окружении, в котором они выполняются. Это может быть информация, известная только во время выполнения, такая как имя пода, IP-адрес и имя хоста, на котором размещено приложение. Или другая статическая информация, определяемая на уровне пода, такая как конкретные запросы и лимиты ресурсов, или динамическая, такая как аннотации и метки, которые могут изменяться пользователем во время выполнения.

Например, в зависимости от ресурсов, доступных контейнеру, приложение может настраивать размер пула потоков выполнения, изменять алгоритм сбора мусора или распределение памяти. Имя пода и имя хоста могут пригодиться для журналирования или отправки метрик на центральный сервер. Также вам может понадобиться отыскать другие поды в том же пространстве имен с определенной меткой и объединить их в кластерное приложение. Для этих и других случаев Kubernetes предоставляет *Downward API*.

Решение

Описанные требования и решение, представленное ниже, относятся не только к контейнерам, но и к любому динамическому окружению, где метаданные ресурсов могут изменяться. Например, AWS предлагает службы Instance Metadata и User Data, которые можно запросить у любого экземпляра EC2. Аналогично, AWS ECS предоставляет API, с помощью которого контейнеры могут запрашивать и получать информацию о кластере.

В Kubernetes используется еще более элегантный и простой подход. *Downward API* позволяет передавать метаданные о поде в контейнеры и в кластер через переменные окружения и файлы. Это те же механизмы, которые мы использовали для передачи данных, связанных с приложением, из ConfigMap и Secret. Но в этом случае данные создаются не нами. Мы просто указываем ключи, которые нас интересуют, а Kubernetes присваивает им значения динамически. На рис. 13.1 представлена общая схема, как Downward API внедряет в поды информацию о ресурсах и о среде времени выполнения.

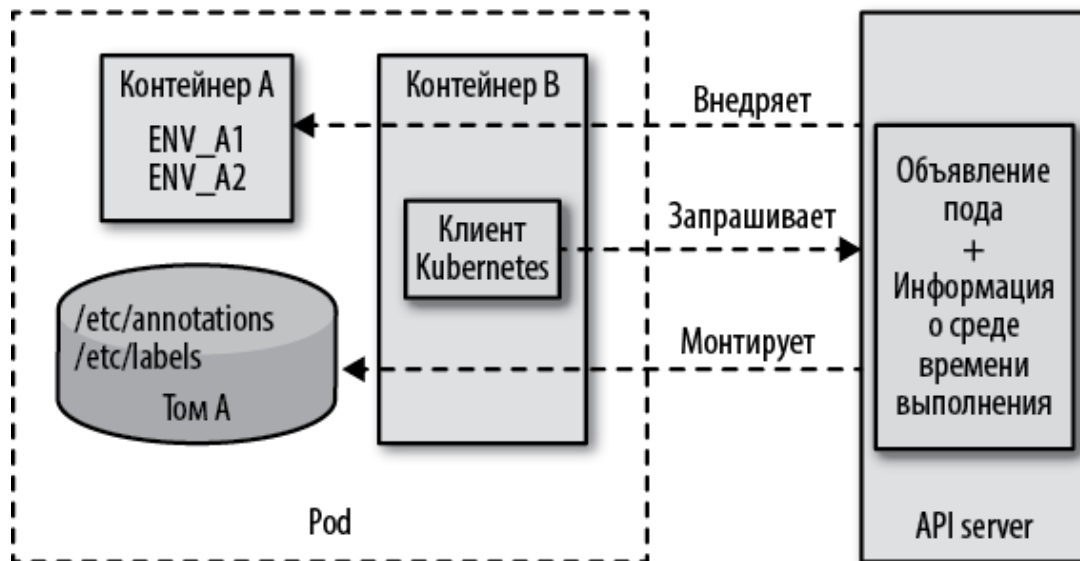


Рис. 13.1. Механизмы интроспекции приложений

Суть в том, что Downward API внедряет метаданные в под и они становятся доступными локально. Приложению не нужно использовать клиента и взаимодействовать с Kubernetes API, и оно может оставаться независимым от Kubernetes. Давайте рассмотрим листинг 13.1, демонстрирующий, насколько просто запрашивать метаданные через переменные окружения.

Листинг 13.1. Передача информации из Downward API через переменные окружения

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    env:
    - name: POD_IP
      valueFrom:
        fieldRef: ❶
          fieldPath: status.podIP
    - name: MEMORY_LIMIT
      valueFrom:
        resourceFieldRef:
          container: random-generator ❷
    resource: limits.memory
```

❶ Переменная окружения `POD_IP` получает значение из свойств пода и инициализируется во время запуска пода.

❷ Переменная окружения `MEMORY_LIMIT` получает значение, равное лимиту памяти, выделенной для этого

контейнера; фактическое объявление лимита здесь не показано.

Для доступа к метаданным уровня пода в этом примере используется свойство `fieldRef`. Ключи, перечисленные в табл. 13.1, доступны в `fieldRef.fieldPath` и в виде переменных окружения, и в виде томов `downwardAPI`.

По аналогии с `fieldRef`, для доступа к метаданным, относящимся к конкретному контейнеру в поде, можно использовать `resourceFieldRef`. Эти метаданные относятся к контейнеру, который можно задать с помощью `resourceFieldRef.container`. При использовании переменной окружения по умолчанию сообщаются метаданные для текущего контейнера. Возможные ключи для `resourceFieldRef.resource` перечислены в табл. 13.2.

Таблица 13.1. Информация из Downward API, доступная в `fieldRef.fieldPath`

Имя	Описание
<code>spec.nodeName</code>	Имя узла, где размещается под
<code>status.hostIP</code>	IP-адрес узла, где размещается под
<code>metadata.name</code>	Имя пода
<code>metadata.namespace</code>	Пространство имен, в котором действует под
<code>status.podIP</code>	IP-адрес пода
<code>spec.serviceAccountName</code>	Учетная запись ServiceAccount, используемая подом
<code>metadata.uid</code>	Уникальный идентификатор пода
<code>metadata.labels['key']</code>	Значение метки key пода
<code>metadata.annotations['key']</code>	Значение аннотации key пода

Таблица 13.2. Информация из Downward API, доступная в resourceFieldRef.resource

Имя	Описание
requests.cpu	Число процессоров, запрошенное контейнером
limits.cpu	Лимит процессоров для контейнера
requests.memory	Объем памяти, запрошенный контейнером
limits.memory	Лимит памяти, доступной контейнеру

Некоторые метаданные, такие как метки и аннотации, могут меняться пользователем во время работы пода. Переменные окружения будут отражать такие изменения только после перезапуска пода. Но тома downwardAPI способны динамически отражать изменения в метках и аннотациях. В дополнение к отдельным полям, перечисленным выше, тома downwardAPI могут хранить все метки и аннотации подов в файлах с именами `metadata.labels` и `metadata.annotations`. Листинг 13.2 показывает, как можно использовать такие тома.

Листинг 13.2. Передача информации из Downward API через тома

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - name: pod-info
      mountPath: /pod-info
  volumes:
```

```

- name: pod-info ❶
  downwardAPI:
    items:
      - path: labels ❷
        fieldRef:
          fieldPath: metadata.labels
      - path: annotations ❸
        fieldRef:
          fieldPath: metadata.annotations

```

❶ Значения из Downward API можно монтировать как файлы в поде.

❷ Файл labels содержит все метки, по одной в каждой строке, в формате имя=значение. Этот файл обновляется синхронно с изменением меток.

❸ Файл annotations содержит все аннотации в том же формате, что и метки.

Если метаданные изменяются во время работы пода, эти изменения синхронно отражаются на содержимом файлов томов. Но приложение-потребитель само должно определять факт изменения файлов и читать обновленные данные. Если такая возможность в приложении не реализована, может потребоваться перезапустить под.

Пояснение

Во многих случаях приложению необходимо иметь информацию о себе и об окружении, в котором оно работает. Kubernetes предлагает ненавязчивые механизмы для интроспекции и внедрения метаданных. Один из недостатков Downward API — ограниченное количество ключей, на которые можно ссылаться. Если приложению нужно больше данных,

особенно о других ресурсах или метаданных, связанных с кластером, оно должно запрашивать их через API Server.

Этот метод используется многими приложениями, которые запрашивают API Server с целью обнаружения других подов в том же пространстве имен, которые имеют определенные метки или аннотации. Затем приложение может сформировать кластер с обнаруженными подами и, например, синхронизировать состояние. Он также используется приложениями мониторинга для поиска интересующих их подов и управления ими.

Существует множество клиентских библиотек на разных языках программирования, предназначенных для взаимодействия с Kubernetes API Server и получения информации о самом приложении, которую нельзя получить с помощью Downward API.

Дополнительная информация

- Пример самоанализа (<http://bit.ly/2TYBXpc>).
- Получение информации контейнером о поде через файлы (<http://bit.ly/2CoZyFy>).
- Получение информации контейнером о поде через переменные окружения (<http://bit.ly/2JpuHPe>).
- Агент интроспекции в Amazon ECS (<https://amzn.to/2JnVXgX>).
- Службы Instance Metadata и User Data (<http://amzn.to/1Cu0fTl>).

Часть III. Структурные паттерны

Образы контейнеров и контейнеры можно сравнить с классами и объектами в объектно-ориентированном мире. Образы контейнеров — это образцы, по которым создаются экземпляры контейнеров. Но контейнеры не работают по отдельности; они объединяются в абстракции более высокого уровня — поды (группы контейнеров), которые предоставляют уникальные возможности времени выполнения.

Паттерны из этой категории ориентированы на структурирование и организацию контейнеров в поды с учетом разных вариантов использования. Силы, которые влияют на контейнеры в подах, приводят к паттернам, обсуждаемым в следующих главах:

- В главе 14 «Init-контейнеры» описывается отдельный жизненный цикл задач, связанных с инициализацией, и основных контейнеров приложений.
- В главе 15 «Паттерн Sidecar» рассказывается, как дополнить и расширить возможности существовавшего контейнера без его изменения.
- В главе 16 «Адаптер» демонстрируется, как гетерогенную систему привести в соответствие единому унифицированному интерфейсу, который может использоваться внешним миром.
- В главе 17 «Посредник» описывается прием организации доступа к внешним службам через прокси.

Глава 14. Init-контейнеры

Паттерн *Init Containers* (Init-контейнеры) позволяет разделить задачи и организовать жизненный цикл для задач, связанных с инициализацией, отдельный от основных контейнеров приложения. В этой главе мы подробно рассмотрим эту основополагающую идею Kubernetes, которая используется во многих других паттернах, когда требуется определить отдельную логику инициализации.

Задача

Задача инициализации широко распространена во многих языках программирования. В некоторых случаях она является частью языка, а в некоторых для обозначения конструкций инициализации используются соглашения об именах и паттерны. Например, в Java, чтобы создать экземпляр объекта, который требует дополнительной настройки, используется конструктор (или, в некоторых необычных ситуациях, статические блоки). Конструкторы гарантированно запускаются средой выполнения на этапе создания экземпляра объекта и только один раз (это только пример; мы не будем вдаваться в подробности, характерные для разных языков и ситуаций). Кроме того, конструктор можно использовать для проверки предварительных условий, например наличия обязательных параметров. Также конструкторы могут использоваться для инициализации полей экземпляра значениями входных аргументов или значениями по умолчанию.

Контейнеры инициализации (или Init-контейнеры) представляют аналогичную абстракцию, но на уровне подов.

То есть если в поде имеется один или несколько контейнеров, которые образуют основное приложение, эти контейнеры могут иметь предварительные условия, которые должны удовлетворяться перед их запуском. К числу таких условий можно отнести, например, настройку специальных разрешений для файловой системы, схемы базы данных или начальных данных приложения. Также логика инициализации может потребовать использовать инструменты и библиотеки, которые нельзя включить в образ приложения. По соображениям безопасности образ приложения может не иметь разрешений для выполнения действий по инициализации. Как вариант, вы можете отложить запуск приложения, пока не будут удовлетворены внешние зависимости. Во всех таких случаях Kubernetes предлагает использовать `init`-контейнеры, чтобы отделить действия, связанные с инициализацией, от основных обязанностей приложения.

Решение

Паттерн *Init Containers* (`Init`-контейнеры) в Kubernetes является частью определения пода и делит все контейнеры в поде на две группы: собственно `init`-контейнеры и контейнеры приложения. Все `init`-контейнеры выполняются последовательно, друг за другом, и все они должны завершиться с признаком успеха, прежде чем Kubernetes приступит к запуску контейнеров приложения. В этом смысле `init`-контейнеры похожи на конструкторы классов в Java, которые помогают инициализировать объекты. Контейнеры приложения, напротив, запускаются параллельно, в произвольном порядке. Описанный процесс показан на рис. 14.1.

Как правило, `init`-контейнеры имеют небольшой размер, быстро запускаются и быстро завершаются с признаком успеха, кроме случаев, когда они используются для задержки запуска подов, пока не будут удовлетворены все необходимые зависимости. В этом последнем случае они могут не завершаться довольно долго. Если `init`-контейнер завершится с признаком ошибки, будет произведен перезапуск всего пода (если только он не отмечен меткой `RestartNever`), в результате чего все `init`-контейнеры будут запущены снова. Поэтому для предотвращения побочных эффектов желательно создавать идемпотентные `init`-контейнеры.

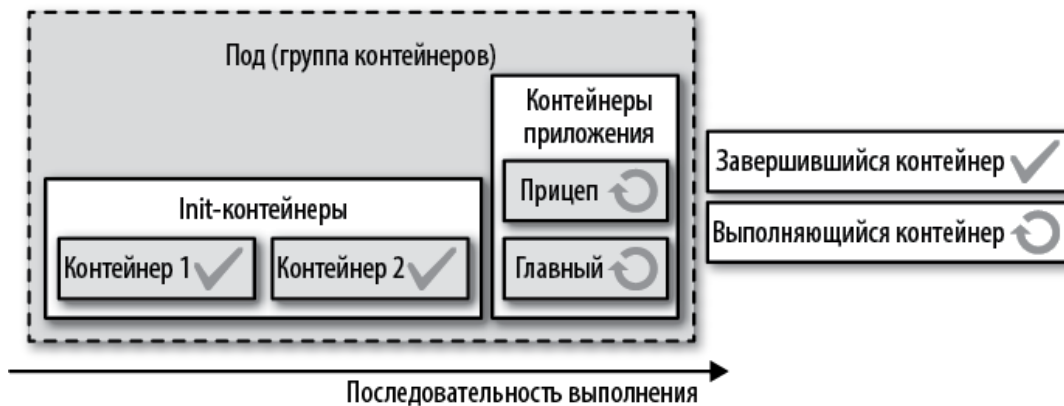


Рис. 14.1. `Init`-контейнеры и контейнеры приложения в поде

С одной стороны, `init`-контейнеры обладают теми же возможностями, что и контейнеры приложений: все контейнеры являются частью одного пода, поэтому для них действуют те же ограничения ресурсов, тома и параметры безопасности и, наконец, они размещаются на одном узле. С другой стороны, они имеют немного другую семантику проверки работоспособности и использования ресурсов. Для `init`-контейнеров не производится проверка работоспособности, так как все эти контейнеры должны успешно завершиться до того, как процесс запуска пода сможет продолжить работу с контейнерами приложения.

Init-контейнеры также влияют на результаты расчетов требований к ресурсам для пода при планировании, автоматическом масштабировании и управлении квотами. Учитывая, что контейнеры в поде выполняются по порядку (сначала последовательно запускаются init-контейнеры, а затем параллельно запускаются все контейнеры приложения), в качестве действующих значений для запросов и лимитов выбираются наибольшие из следующих двух групп:

- наибольшее значение запроса/лимита для init-контейнера;
- сумма всех значений запроса/лимита для контейнера приложения.

Вследствие этого, если имеются init-контейнеры с высокими требованиями и контейнеры приложений с низкими требованиями к ресурсам, значения запросов и лимитов на уровне пода, влияющие на планирование, будут вычисляться на основе высоких значений для init-контейнеров. Это не оптимальное решение с точки зрения эффективного расходования ресурсов. Даже если init-контейнеры работают очень короткое время и в большинстве случаев на узле остается свободным некоторый объем ресурсов, никакой другой под не сможет их использовать.

Кроме того, init-контейнеры позволяют разделить задачи и сделать контейнеры более специализированными. Прикладной программист может создавать контейнеры, сосредоточенные только на логике приложений. Инженер по развертыванию может создавать init-контейнеры, сосредоточенные только на задачах настройки и инициализации. Мы продемонстрируем это в листинге 14.1, где определяется один контейнер приложения на основе HTTP-сервера, обслуживающий файлы.

Контейнер реализует универсальную функцию обслуживания HTTP-клиентов и не делает никаких предположений о том, откуда берутся файлы для разных клиентов. Init-контейнер в том же поде реализует клиента Git с единственной целью клонирования репозитория. Поскольку оба контейнера являются частью одного пода, они имеют доступ к одному и тому же тому для хранения общих данных. Мы используем тот же механизм для передачи клонированных файлов из init-контейнера в контейнер приложения.

Листинг 14.1 содержит определение init-контейнера, который копирует данные в пустой том.

Листинг 14.1. Init-контейнер

```
apiVersion: v1
kind: Pod
metadata:
  name: www
  labels:
    app: www
spec:
  initContainers:
  - name: download
    image: axec1br/git
    command:
      - git
      - clone
      - https://github.com/mdn/beginner-html-
site-scripted
    - /var/lib/data
  volumeMounts:
  - mountPath: /var/lib/data
    name: source
```

❶

❷

```
containers:
- name: run
  image: docker.io/centos/httpd
  ports:
  - containerPort: 80
  volumeMounts: ②
  - mountPath: /var/www/html
    name: source
volumes: ③
- emptyDir: {}
  name: source
```

① Клонирование внешнего репозитория Git в смонтированный каталог.

② Общий том, используемый обоими контейнерами.

③ Пустой каталог на узле, используемый для общих данных.

Того же эффекта можно добиться с помощью ConfigMap или PersistentVolume, но здесь я хотел показать, как работают init-контейнеры. Этот пример иллюстрирует типичный подход к совместному использованию тома init-контейнером и основным контейнером приложения.



Задержка запуска пода

Для отладки результатов init-контейнеров бывает полезно временно заменить запуск контейнера приложения фиктивной командой `sleep`, чтобы получить время на изучение ситуации.

Этот прием может особенно пригодиться, если init-контейнер терпит неудачу и приложение не запускается из-за отсутствия должной конфигурации окружения. Следующая команда в определении пода дает вам час для отладки смонтированных томов после входа в под с помощью команды `kubectl exec -it <под> sh:`

`command:`

- `/bin/sh`
- `"-c"`
- `"sleep 3600"`

Другие приемы инициализации

Как вы уже видели, init-контейнер – это конструкция уровня пода, которая активируется после его запуска. В Kubernetes есть еще несколько методов, используемых для инициализации ресурсов, не связанных с init-контейнерами, и я перечислю их для полноты картины:

Контроллеры доступа

Это набор плагинов, которые перехватывают все запросы, поступающие в Kubernetes API Server перед сохранением объекта, и могут изменять или проверять его. Существует множество контроллеров, которые выполняют разные проверки, применяют ограничения и настраивают значения по умолчанию, но все они скомпилированы в двоичный файл `kube-apiserver` и настраиваются администратором кластера при запуске API Server. Система плагинов не

отличается гибкостью, поэтому в Kubernetes были добавлены веб-обработчики доступа.

Веб-обработчики доступа

Эти компоненты являются внешними контроллерами доступа, которые выполняют обратные вызовы HTTP для любого соответствующего запроса. Существует два типа веб-обработчиков доступа: *изменяющие* (способные изменять ресурсы для применения пользовательских настроек) и *проверяющие* (способные отклонять ресурсы в соответствии с пользовательскими политиками доступа). Идея внешних контроллеров позволяет разрабатывать веб-обработчики доступа вне Kubernetes и настраивать их во время выполнения.

Инициализаторы

Инициализаторы могут использоваться администраторами для принудительного применения политик или настройки значений по умолчанию путем определения списка задач предварительной инициализации, имеющегося в метаданных любого объекта. После определения такого списка пользовательские контроллеры инициализации, имена которых соответствуют именам задач, будут выполнять эти задачи. И только после завершения всех задач инициализации объект API становится видимым для обычных контроллеров.

Предварительные настройки PodPreset

Предварительные настройки PodPreset применяются еще одним контроллером доступа, который помогает внедрять в поды поля, указанные в соответствующих наборах предварительных настроек PodPreset, во время их создания. Поля могут включать тома, монтируемые тома или переменные окружения. То есть PodPreset вводит дополнительные требования к окружению времени выполнения в подах на этапе их создания, используя селекторы меток для выбора подов, к которым применяется данный набор настроек PodPreset. Наборы PodPreset позволяют авторам паттернов подов автоматизировать добавление повторяющейся информации, необходимой нескольким подам.

Существует много способов инициализации ресурсов в Kubernetes. Но они отличаются от веб-обработчиков доступа тем, что проверяют и изменяют ресурсы во время создания. Эти методы можно использовать, например, чтобы вставить `init`-контейнер в любой под, где такого контейнера еще нет. Паттерн `Init Container` (`Init-контейнер`), обсуждаемый в этой главе, напротив, активируется и выполняет свою работу во время запуска пода. Но самое существенное отличие – `init`-контейнеры предназначены для разработчиков, развертывающих свои приложения в Kubernetes, а описанные здесь методы помогают администраторам контролировать и управлять процессом инициализации контейнеров.

Похожий эффект можно получить при использовании паттерна *Sidecar* (Прицеп), описанного далее в главе 15, где

контейнер с HTTP-сервером и контейнер с клиентом Git работают вместе, как контейнеры приложения. Но паттерн *Sidecar* (Прицеп) не позволяет узнать, какой контейнер запустится первым, и предназначен для случаев, когда контейнеры постоянно работают вместе (как в листинге 15.1, где контейнер синхронизации с репозиторием Git постоянно обновляет содержимое локальной папки). Паттерны *Sidecar* (Прицеп) и *Init Container* (Init-контейнер) также можно использовать в паре, если требуются гарантированная инициализация и постоянное обновление данных.

Пояснение

Итак, зачем разделять контейнеры в подах на две группы? Почему бы просто не использовать обычный контейнер приложения для инициализации пода, если это необходимо? Дело в том, что эти две категории контейнеров имеют разные жизненные циклы, цели и иногда даже авторов.

Init-контейнеры запускаются перед контейнерами приложений, и, что особенно важно, init-контейнеры запускаются последовательно и только после успешного завершения текущего init-контейнера. То есть на каждом этапе процедуры инициализации можно быть уверенными, что предыдущий этап успешно завершен и можно смело переходить к следующему этапу. Контейнеры приложений, напротив, запускаются параллельно и не дают гарантий, которые дают init-контейнеры. Благодаря этим различиям можно создавать контейнеры, ориентированные на инициализацию или решение прикладных задач, и организовывать их в поды.

Дополнительная информация

- Пример init-контейнера (<http://bit.ly/2TW7ckN>).
- Init-контейнеры (<http://bit.ly/2TR7OsD>).
- Настройка инициализации пода (<http://bit.ly/2TWMEbL>).
- Паттерн Initializer (Инициализатор) в JavaScript (<http://bit.ly/2TYF14G>).
- Инициализация объектов в Swift (<https://apple.co/2FdSLPN>).
- Использование контроллеров доступа (<http://bit.ly/2ztKrJM>).
- Динамические контроллеры доступа (<http://bit.ly/2DwR2Y3>).
- Как работают инициализаторы в Kubernetes (<http://bit.ly/2TeYz0k>).
- Наборы предварительных настроек подов PodPreset (<https://kubernetes.io/docs/concepts/workloads/pods/podpreset/>)
.
- Внедрение информации в поды с помощью PodPreset (<http://bit.ly/2Fh7QzV>).
- Руководство по инициализаторам в Kubernetes (<http://bit.ly/2FfEu4W>).

Глава 15. Паттерн *Sidecar*

Паттерн *Sidecar* (Прицеп) заключается в определении контейнера, который расширяет возможности существующего контейнера без его изменения. Это один из основополагающих паттернов контейнеров, который позволяет создавать узкоспециализированные контейнеры, тесно взаимодействующие друг с другом. В этой главе вы узнаете все, что связано с идеей паттерна *Sidecar* (Прицеп). А в главах 16 и 17 вы познакомитесь со специализированными вариантами этого паттерна — паттернами *Adapter* (Адаптер) и *Ambassador* (Посредник) соответственно.

Задача

Контейнеры — популярная технология упаковки, которая позволяет разработчикам и системным администраторам создавать, доставлять и запускать приложения унифицированным способом. Контейнер представляет естественную границу функциональной единицы со своей средой времени выполнения, циклом выпуска, API и коллективом разработчиков, которому он принадлежит. Типичный контейнер действует подобно процессу в Linux — решает одну проблему, и делает это хорошо, — и создается в предположении возможности замены и повторного использования. Последнее особенно важно, поскольку позволяет быстрее создавать приложения с использованием существующих специализированных контейнеров.

В настоящее время, чтобы послать HTTP-запрос, не нужно писать клиентскую библиотеку, достаточно использовать уже существующую. Аналогично, для обслуживания веб-сайта не

нужно создавать контейнер с веб-сервером, достаточно использовать уже существующий. Этот подход позволяет разработчикам не изобретать колесо и создать экосистему с меньшим количеством контейнеров лучшего качества для обслуживания. Однако чтобы иметь возможность использовать узкоспециализированные многоразовые контейнеры, необходимы способы расширения их возможностей и средства для организации взаимодействий между ними. Паттерн *Sidecar* (Прицеп) описывает как раз такой способ организации взаимодействий, когда один контейнер расширяет возможности другого, уже существующего контейнера.

Решение

В главе 1 мы видели, как поды позволяют объединить несколько контейнеров в один блок. За кулисами, во время выполнения, под также является контейнером, но запускается как приостановленный (буквально с помощью команды `pause`) процесс перед всеми остальными контейнерами в поде. Он не делает ничего, кроме того, что хранит все пространства имен Linux, которые контейнеры приложений используют для взаимодействия на протяжении жизненного цикла пода. Кроме этой детали реализации, интерес представляют также все характеристики, которые предоставляет абстракция пода.

Под является настолько фундаментальным примитивом, что присутствует во многих облачных платформах, пусть под разными именами, но всегда со схожими возможностями. Под, как единица развертывания, накладывает определенные ограничения времени выполнения на принадлежащие ему контейнеры. Например, все контейнеры разворачиваются на одном узле и имеют общий жизненный цикл. Кроме того, под позволяет своим контейнерам использовать общие тома и

обмениваться данными через локальную сеть или с применением средств межпроцессных взаимодействий хоста. Именно поэтому пользователи объединяют контейнеры в поды. Паттерн *Sidecar* (Прицеп), который иногда называют также *Sidekick* (Компаньон), описывает сценарий добавления контейнера в под для расширения возможностей другого контейнера.

Типичным примером, демонстрирующим этот паттерн, может служить HTTP-сервер и механизм синхронизации с репозиторием Git. Контейнер HTTP-сервера решает задачи, связанные с обслуживанием файлов через HTTP и не знает, как и откуда эти файлы поступают. Точно так же единственной целью контейнера, осуществляющего синхронизацию с Git, является синхронизация данных в локальной файловой системе с данными на сервере Git. Ему неважно, что происходит с файлами после синхронизации, его единственная задача — синхронизировать содержимое локальной папки с содержимым на удаленном сервере Git. В листинге 15.1 приводится определение пода с этими двумя контейнерами, настроенными на использование тома для обмена файлами.

Листинг 15.1. Реализация паттерна Sidecar (Прицеп)

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: app
    image: docker.io/centos/httpd ❶
    ports:
    - containerPort: 80
  volumeMounts:
```

```

    - mountPath: /var/www/html           ❸
      name: git
- name: poll
  image: axecldr/git                     ❷
  volumeMounts:
    - mountPath: /var/lib/data          ❸
      name: git
  env:
    - name: GIT_REPO
      value: https://github.com/mdn/beginner-
html-site-scripted
  command:
    - "sh"
    - "-c"
    - "git clone $(GIT_REPO) . && watch -n 600
git pull"
  workingDir: /var/lib/data
  volumes:
    - emptyDir: {}
      name: git

```

❶ Основной контейнер приложения, обслуживающий файлы через HTTP.

❷ Вспомогательный контейнер (Прицеп), действующий параллельно и получающий данные с сервера Git.

❸ Общая папка для обмена данными между основным и вспомогательным контейнерами.

Этот пример показывает, как контейнер синхронизации с Git добавляет контент для обслуживания сервером HTTP и поддерживает его в актуальном состоянии. Можно также сказать, что оба контейнера действуют в тесном

сотрудничестве и одинаково важны, но паттерн *Sidecar* (Прицеп) предполагает наличие основного (главного) и вспомогательного контейнера, который расширяет коллективное поведение. Обычно главный перечисляется в списке контейнеров первым и представляет контейнер по умолчанию (который запускается командой `kubectl exec`, например).

Этот простой паттерн, изображенный на рис. 15.1, обеспечивает тесное сотрудничество контейнеров во время выполнения и в то же время позволяет разделить задачи, которые могут принадлежать отдельным коллективам разработчиков, использующим разные языки программирования,

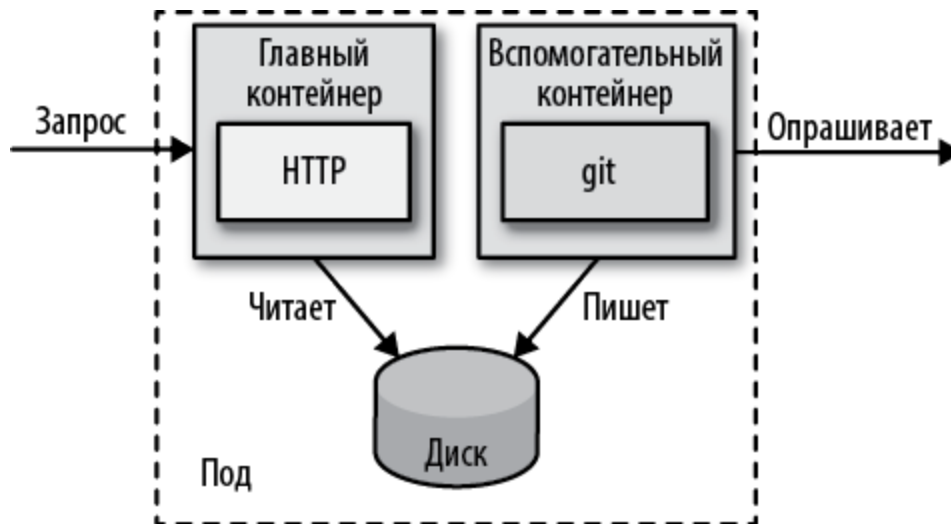


Рис. 15.1. Паттерн Sidecar (Прицеп)

с разными циклами выпуска новых версий и т.д. Он также способствует взаимозаменяемости и многократному использованию контейнеров — в том смысле, что HTTP-сервер и механизм синхронизации Git могут повторно использоваться в других приложениях и с другими настройками, как самостоятельно, так и совместно с другими контейнерами.

Пояснение

Выше уже говорилось, что образы контейнеров подобны классам, а контейнеры — объектам в объектно-ориентированном программировании (ООП). Продолжая эту аналогию, можно сравнить расширение возможностей контейнера с наследованием в ООП, а совместную работу нескольких контейнеров в поде — с приемом композиции в ООП. Оба этих подхода допускают повторное использование кода, но наследование определяет более тесную связь и представляет отношение «является» между контейнерами.

Композиция в поде представляет отношение «имеет» — более гибкое, потому что не требует объединения контейнеров во время сборки, что дает возможность потом поменять контейнеры в определении пода. Но композиция также подразумевает наличие нескольких контейнеров (процессов), действующих одновременно, которые нужно проверять на работоспособность и перезапускать и которые потребляют ресурсы, как и главный контейнер приложения. Паттерн *Sidecar* (Прицеп) предполагает создание маленьких вспомогательных контейнеров, потребляющих минимальные ресурсы, но вам решать, стоит ли запускать отдельный процесс или лучше объединить все задачи в основной контейнер.

С другой точки зрения композиция контейнеров похожа на аспектно-ориентированное программирование, когда с помощью дополнительных контейнеров в под добавляются ортогональные (независимые) возможности, не касающиеся главного контейнера. В последние месяцы паттерн *Sidecar* (Прицеп) обретает все большую популярность, особенно для решения задач управления сетью, мониторинга служб, где каждая служба также поставляется в виде вспомогательных контейнеров.

Дополнительная информация

- Пример реализации паттерна *Sidecar* (Прицеп, <http://bit.ly/2FqSZUV>).
- Паттерны проектирования для распределенных систем на основе контейнеров (<http://bit.ly/2Odan24>).
- Prana: вспомогательный контейнер для приложений и служб с поддержкой Netflix на основе PaaS (<http://bit.ly/2Y9PRnS>).
- Корабельный телефон: паттерны для добавления авторизации и шифрования в устаревшие приложения (<https://www.feval.ca/posts/tincan-phone/>).
- Универсальный контейнер для приостановки (<http://bit.ly/2FOYH21>).

Глава 16. Адаптер

Паттерн *Adapter* (Адаптер) позволяет привести гетерогенную контейнерную систему в соответствие с единообразным унифицированным интерфейсом, имеющим стандартизованный и нормализованный формат, который может использоваться внешним миром. Паттерн *Adapter* (Адаптер) наследует все свои характеристики от паттерна *Sidecar* (Прицеп), но имеет единственную цель — предоставить адаптированный доступ к приложению.

Задача

Контейнеры позволяют унифицировать упаковку и запуск приложений, написанных на разных языках и с использованием разных библиотек. В настоящее время многие команды используют разные технологии и создают распределенные системы, состоящие из гетерогенных (разнородных) компонентов. Эта разнородность может вызвать трудности, когда все компоненты должны обрабатываться другими системами единообразным способом. Паттерн *Adapter* (Адаптер) предлагает решение, помогающее скрыть сложность системы и предоставить унифицированный доступ к ней.

Решение

Этот паттерн лучше проиллюстрировать на примере. Основным условием успешной эксплуатации распределенных систем является всеобъемлющий мониторинг и возможность отправки оповещений. Кроме того, если распределенная система состоит из нескольких служб, для их мониторинга

можно использовать внешний инструмент, извлекающий и сохраняющий метрики каждой службы.

Однако службы, написанные на разных языках, могут иметь разные возможности и поставлять метрики в разных форматах, отличных от ожидаемого инструментом мониторинга. Такое разнообразие создает проблему для мониторинга разнородных приложений с использованием единого решения, которое ожидает поддержки единого формата всей системой. Паттерн *Adapter* (Адаптер) позволяет организовать унифицированный интерфейс мониторинга, экспортируя метрики из различных контейнеров приложений в один стандартный формат и протокол. На рис. 16.1 изображен контейнер-адаптер, который преобразует информацию с метриками, хранящуюся локально, во внешний формат, который понимает сервер мониторинга.

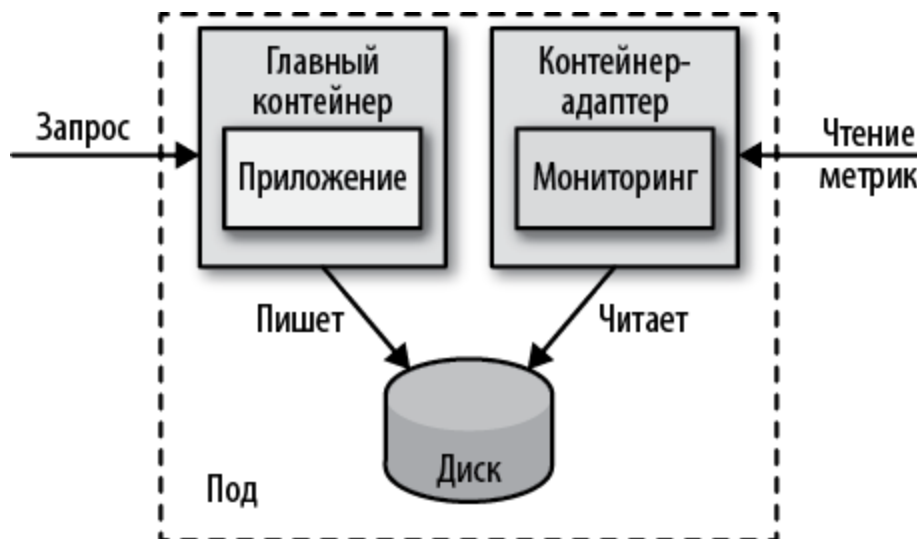


Рис. 16.1. Паттерн Adapter (Адаптер)

При таком подходе каждая служба, представленная отдельным подом, кроме основного контейнера приложения будет включать еще один контейнер, который знает, как прочитать метрики приложения и преобразовать их в формат, понятный инструменту мониторинга. Например, мы можем

иметь один контейнер-адаптер, который знает, как экспортировать Java-метрики через HTTP, и другой контейнер-адаптер, в другом поде, который знает, как экспортировать Python-метрики через HTTP. Инструменту мониторинга все метрики будут доступны через HTTP в общем нормализованном формате.

Чтобы познакомиться с конкретной реализацией этого паттерна, вернемся к нашему примеру приложения генератора случайных чисел и создадим адаптер, действующий, как показано на рис. 16.1. При соответствующей настройке он читает файл журнала генератора случайных чисел и извлекает из него время создания случайного числа. Предполагается, что мы должны контролировать это время с помощью Prometheus. К сожалению, формат журнала не соответствует формату, который ожидает Prometheus. Кроме того, мы должны открыть доступ к этой информации через конечную точку HTTP, чтобы сервер Prometheus мог получить ее.

Паттерн *Adapter* (Адаптер) идеально подходит для этого случая: вспомогательный контейнер запускает небольшой HTTP-сервер и при каждом запросе читает файл журнала и преобразует его в формат, понятный для Prometheus. В листинге 16.1 представлено определение развертывания Deployment с таким контейнером-адаптером. Эта конфигурация избавляет основное приложение от необходимости знать что-либо о Prometheus. Полный пример в нашем репозитории GitHub демонстрирует эту конфигурацию вместе с установкой Prometheus.

Листинг 16.1. Адаптер, преобразующий информацию в формат, понятный Prometheus

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: random-generator
```

```

spec:
  replicas: 1
  selector:
    matchLabels:
      app: random-generator
  template:
    metadata:
      labels:
        app: random-generator
    spec:
      containers:
        - image: k8spatterns/random-
generator:1.0 ❶
          name: random-generator
          env:
            - name:
LOG_FILE ❷
              value: /logs/random.log
          ports:
            - containerPort: 8080
              protocol: TCP
          volumeMounts:
            ❸
            - mountPath: /logs
              name: log-volume
          # -----
          -----
            - image: k8spatterns/random-generator-
exporter ❹
              name: prometheus-adapter
          env:

```

```

LOG_FILE
    value: /logs/random.log
    ports:
      - containerPort: 9889
        protocol: TCP
    volumeMounts:
      ⑥
      - mountPath: /logs
        name: log-volume
    volumes:
      - name: log-
        volume
          ⑦
          emptyDir: {}

```

① Главный контейнер приложения со службой, генерирующей случайные числа, доступной через порт 8080.

② Путь к файлу журнала с информацией о времени создания случайных чисел.

③ Каталог, используемый совместно с контейнером-адаптером.

④ Образ контейнера, экспортирующего информацию в Prometheus через порт 9889.

⑤ Путь к файлу журнала, куда основное приложение записывает интересующую нас информацию.

⑥ Общий том монтируется также в контейнер-адаптер.

⑦ Общие файлы находятся в томе `emptyDir`, в файловой системе узла.

Этот паттерн может также использоваться для журналирования. Разные контейнеры могут записывать информацию в разных форматах и с разной степенью

детализации. Контейнер-адаптер может нормализовать эту информацию, очистить ее, обогатить контекстной информацией с помощью паттерна *Self Awareness* (Самоанализ), описанного в главе 13, а затем сделать результат доступным для централизованного агрегатора журналов.

Пояснение

Паттерн *Adapter* (Адаптер) — это специализированный вариант паттерна *Sidecar* (Прицеп), описанного в главе 15. Он действует как обратный прокси для гетерогенной системы, скрывая ее сложность за унифицированным интерфейсом. Использование отдельного названия, отличного от названия более общего паттерна *Sidecar* (Прицеп), позволяет более точно обозначить назначение этого паттерна.

В следующей главе вы познакомитесь с еще одним специализированным вариантом паттерна *Sidecar* (Прицеп): паттерн *Ambassador* (Посредник), который действует как прокси, для доступа к службам во внешнем мире.

Дополнительная информация

- Пример реализации паттерна Адаптер (<http://bit.ly/2HvFF3Y>).
- Инструменты распределенных систем: паттерны контейнеров для создания модульных распределенных систем (<https://bit.ly/2U2iWD9>).

Глава 17. Посредник

Паттерн *Ambassador* (Посредник) — это специализированный вариант паттерна *Sidecar* (Прицеп), отвечающий за сокрытие сложности и предоставляющий унифицированный интерфейс для доступа к службам за пределами пода. В этой главе мы увидим, как паттерн *Ambassador* (Посредник) может выступать в качестве прокси и ограждать поды от прямого доступа к внешним зависимостям.

Задача

Контейнерные службы существуют не в вакууме и часто обращаются к другим службам, надежный доступ к которым может быть затруднен. Сложность доступа к другим службам может быть обусловлена динамическими и изменяющимися адресами, необходимостью балансировки нагрузки между кластерными экземплярами служб, использованием ненадежного протокола или сложных форматов данных. В идеале контейнеры должны быть узкоспециализированными и допускающими возможность многократного использования. Но если у нас есть контейнер, который реализует некоторую важную функцию и особым образом использует внешнюю службу, он будет нести больше одной ответственности.

Для доступа к внешней службе может потребоваться использовать специальную библиотеку обнаружения служб, которую нежелательно помещать в контейнер. Или может понадобиться заменить одни службы другими и использовать другие библиотеки и методы обнаружения служб. Абстрагирование и изоляция логики доступа к другим службам

во внешнем мире как раз и являются целью паттерна *Ambassador* (Посредник).

Решение

Продемонстрируем использование паттерна на примере организации кэша для приложения. Для доступа к локальному кэшу в окружении для разработки может быть достаточно определить некоторые настройки в конфигурации, но в промышленном окружении может понадобиться настроить клиент, подключающийся к разным сегментам кэша. Другой пример — поиск службы в реестре для ее обнаружения на стороне клиента. Еще один пример — взаимодействие с внешней службой по ненадежному протоколу, такому как HTTP, из-за чего для защиты приложения приходится использовать логику обработки обрывов связи, настраивать тайм-ауты, выполнять повторные попытки и многое другое.

Во всех этих случаях можно использовать контейнер-посредник, скрывающий сложность доступа к внешним службам и обеспечивающий упрощенное их представление для основного контейнера приложения через локальное сетевое соединение. На рис. 17.1 и 17.2 показано, как контейнер-посредник помогает отделить прикладную логику от логики доступа к хранилищу пар ключ/значение, принимая запросы через локальный порт. На рис. 17.1 видно, как можно делегировать доступ к данным, хранящимся в удаленном распределенном хранилище, таком как Etcd.



Рис. 17.1. Посредник для доступа к удаленному распределенному кешу

Во время разработки такой контейнер-посредник легко заменить локальным хранилищем пар ключ/значение, таким как memcached (как показано на рис. 17.2).

Этот паттерн предлагает те же преимущества, что и паттерн *Sidcar* (Прицеп) — оба способствуют узкой специализации контейнеров и пригодности их к многократному использованию. При использовании этого

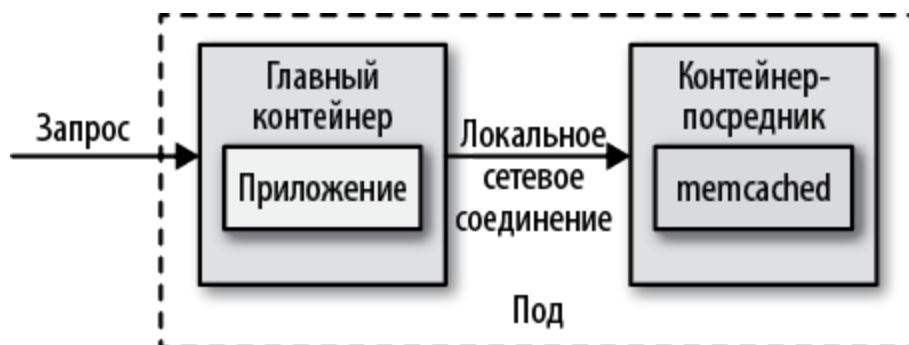


Рис. 17.2. Контейнер-посредник с локальным хранилищем

паттерна приложение может сосредоточиться на решении бизнес-задач и делегировать ответственность за использование внешней службы другому специализированному контейнеру. Это также позволяет создавать специализированные и повторно используемые контейнеры-посредники, которые можно объединять с другими контейнерами приложений.

В листинге 17.1 показано определение посредника, действующего параллельно REST-службе. Прежде чем вернуть ответ, REST-служба журналирует сгенерированные данные,

отправляя их по фиксированному URL: *http://localhost:9009*. Процесс-посредник прослушивает этот порт и обрабатывает данные. В этом примере он просто выводит данные в консоль, но вообще мог бы делать что-то более сложное, например, переслать данные в полноценную инфраструктуру журналирования. Для REST-службы не имеет значения, что происходит с журналируемыми данными, и вы легко можете заменить контейнер-посредник перенастройкой пода, не касаясь основного контейнера.

Листинг 17.1. Посредник для вывода журналируемых данных

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
  labels:
    app: random-generator
spec:
  containers:
    - image: k8spatterns/random-generator:1.0 ❶
      name: main
      env:
        - name: LOG_URL ❷
          value: http://localhost:9009
      ports:
        - containerPort: 8080
          protocol: TCP
        - image: k8spatterns/random-generator-log-ambassador ❸
          name: ambassador
```

❶ Главный контейнер приложения содержит REST-службу, которая генерирует случайные числа.

❷ Адрес URL для подключения к контейнеру-посреднику через локальное соединение.

❸ Контейнер-посредник, выполняющийся параллельно и прослушивающий порт 9009 (который не экспортируется за пределы пода).

Пояснение

Фактически паттерн *Ambassador* (Посредник) — это все тот же паттерн *Sidecar* (Прицеп). Основное отличие между ними заключается в том, что Посредник не добавляет новых возможностей в основное приложение, а просто действует подобно интеллектуальному прокси-серверу, открывающему выход во внешний мир, из-за чего этот паттерн иногда называют *Proxu* (Представитель).

Дополнительная информация

- Пример реализации паттерна Посредник (<http://bit.ly/2FpjBFS>).
- Как использовать паттерн Посредник для организации динамически настраиваемых служб (<https://do.co/2HxGIQG>).
- Динамическое связывание контейнеров Docker с использованием паттерна Посредник (<http://bit.ly/2TQ1uBO>).
- Организация взаимодействий через контейнер-посредник (<https://dockr.ly/2UdTGKc>).

- Варианты базового паттерна Посредник (<http://bit.ly/2Ju4zmb>).

Часть IV. Конфигурационные паттерны

Любое приложение нужно конфигурировать, и самый простой способ сделать это — сохранить конфигурации в исходном коде. У этого подхода есть побочный эффект: конфигурация и код живут и умирают вместе, как описывает Киф Моррис (Kief Morris) в статье «Immutable Server» (<http://bit.ly/2CoH5cj>). Нам же нужна гибкость, позволяющая адаптировать конфигурацию без повторного создания образа приложения. Фактически повторное создание может потребовать много времени и считается антипаттерном в методике непрерывной доставки, когда приложение создается один раз, а затем перемещается без изменений через разные этапы конвейера развертывания, пока не достигнет места промышленной эксплуатации.

Как в таком сценарии можно было бы адаптировать приложение к различным настройкам окружений разработки, тестирования и эксплуатации? Ответ заключается в использовании внешних конфигурационных данных, различных для каждого окружения. Паттерны, которые приводятся в следующих главах, описывают настройку и адаптацию приложений к разным окружениям с использованием внешних конфигурационных данных:

- Глава 18 «Конфигурация в переменных окружения» описывает порядок использования переменных окружения для хранения конфигурационных данных.
- Глава 19 «Конфигурация в ресурсах» описывает использование ресурсов Kubernetes, таких как ConfigMap и Secret, для хранения информации о конфигурации.

- Глава 20 «Неизменяемая конфигурация» описывает способ хранения в неизменном виде больших наборов данных, помещая их в контейнеры, подключаемые к контейнерам приложений во время выполнения.
- Глава 21 «Макет конфигурации» демонстрирует прием, который может пригодиться, когда для хранения настроек разных окружений, имеющих лишь небольшие отличия, используются очень большие конфигурационные файлы.

Глава 18. Конфигурация в переменных окружения

Рассматривая паттерн *EnvVar Configuration* (Конфигурация в переменных окружения), мы познакомимся с самым простым способом настройки приложений. Самый простой способ определить конфигурацию, когда количество настраиваемых параметров невелико, — поместить их в переменные окружения. Мы уже видели разные способы определения переменных окружения в Kubernetes, а также познакомились с некоторыми ограничениями, препятствующими их использованию для сложных конфигураций.

Задача

Всякое нетривиальное приложение требует определения параметров для доступа к источникам данных и внешним службам или для тонкой настройки поведения в промышленном окружении. Еще до появления манифеста «Двенадцать факторов» мы знали, что хранить конфигурацию внутри приложения неправильно. Конфигурацию следует вынести за рамки приложения, чтобы ее можно было изменить даже после его сборки. Это еще больше увеличивает ценность контейнерных приложений, которые позволяют использовать неизменяемые артефакты и способствуют этому. Но как это сделать в контейнерном мире?

Решение

Для хранения конфигураций приложений манифест «Двенадцать факторов» рекомендует использовать переменные

окружения. Этот подход прост и может использоваться в любых окружениях и на любых платформах. Любая операционная система позволяет определять переменные окружения и передавать их в приложениях, и каждый язык программирования поддерживает простые средства доступа к этим переменным. Можно смело утверждать, что переменные окружения — это универсальный механизм. Обычно при использовании переменных окружения во время сборки определяются их значения по умолчанию, а во время выполнения они изменяются. Давайте рассмотрим некоторые конкретные примеры, как это реализовать в Docker и Kubernetes.

В образах Docker переменные окружения можно определить прямо в файлах *Dockerfile*, с помощью директивы ENV, по одной переменной в строке, как показано в листинге 18.1.

Листинг 18.1. Файл Dockerfile с определениями переменных окружения

```
FROM openjdk:11
ENV PATTERN "EnvVar Configuration"
ENV LOG_FILE "/tmp/random.log"
ENV SEED "1349093094"

# Альтернативный вариант:
ENV          PATTERN="EnvVar          Configuration"
LOG_FILE=/tmp/random.log SEED=1349093094
...
```

Приложение на Java, действующее в таком контейнере, может легко получить доступ к переменным с помощью стандартной библиотеки Java, как показано в листинге 18.2.

Листинг 18.2. Чтение переменных окружения в Java

```
public Random initRandom() {
```

```
        long seed =  
Long.parseLong(System.getenv("SEED"));  
    return new Random(seed); ❶  
}
```

❶ Инициализация генератора случайных чисел начальным значением из переменной окружения SEED.

о значениях по умолчанию

Значения по умолчанию упрощают жизнь, потому что избавляют от бремени выбора значения для параметра конфигурации, о существовании которого вы можете даже не подозревать. Они также играют важную роль в парадигме *преобладания соглашений перед настройками*. Однако не всегда желательно определять значения по умолчанию. А иногда такой подход может даже быть антипаттерном.

Причина в том, что ретроспективное *изменение* значений по умолчанию – сложная задача. Во-первых, чтобы изменить значение по умолчанию, нужно изменить код, а в этом случае придется выполнить повторную сборку приложения. Во-вторых, люди, полагающиеся на значения по умолчанию (осознанно или нет), могут быть неприятно удивлены, если эти значения изменятся. Мы должны явно сообщать о таких изменениях, чтобы пользователи нашего приложения смогли изменить вызывающий код.

С другой стороны, значения по умолчанию часто приходится менять, потому что трудно выбрать достаточно хорошие значения с самого начала. Поэтому изменение значений по

умолчанию следует рассматривать как *существенную модификацию*, и если используется семантическое управление версиями, такие изменения должны вызывать увеличение основного номера версии. Если не удастся найти удовлетворительное значение по умолчанию, часто лучше вообще удалить его и генерировать ошибку, если пользователь не определил свое значение параметра. Это по крайней мере явно нарушит работу приложения и не приведет к неожиданным и труднодиагностируемым проблемам.

Учитывая все эти проблемы, часто лучше с самого начала *отказаться от значений по умолчанию*, если вы не уверены хотя бы на 90%, что выбрали разумное значение по умолчанию. Пароли или параметры подключения к базе данных – вот яркие примеры настроек, для которых не должно быть значений по умолчанию, потому что они сильно зависят от окружения и часто не имеют разумных значений. Кроме того, в отсутствие значений по умолчанию мы должны явно представить информацию о настройке, что также послужит дополнительной документацией.

Если запустить этот образ как есть, он будет использовать жестко заданные значения по умолчанию. Но часто бывает желательно переопределить их извне образа.

Сделать это можно, добавив новые значения для переменных окружения в команду, запускающую контейнер Docker, как показано в листинге 18.3.

Листинг 18.3. Переопределение переменных окружения при запуске контейнера Docker

```
docker run -e PATTERN="EnvVarConfiguration" \  
          -e LOG_FILE="/tmp/random.log" \  
          -e SEED="147110834325" \  
          k8spatterns/random-generator:1.0
```

В Kubernetes переменные окружения можно переопределить непосредственно в спецификации пода, в контроллере Deployment или ReplicaSet (как показано в листинге 18.4).

Листинг 18.4. Переопределение переменных окружения в контроллере Deployment

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: random-generator  
spec:  
  containers:  
  - image: k8spatterns/random-generator:1.0  
    name: random-generator  
    env:  
    - name: LOG_FILE  
      value: /tmp/random.log ❶  
    - name: PATTERN  
      valueFrom:  
        configMapKeyRef: ❷  
          name: random-generator-config ❸  
          key: pattern ❹  
    - name: SEED  
      valueFrom:  
        secretKeyRef: ❺  
          name: random-generator-secret  
          key: seed
```

- ❶ Фактическое значение для переменной окружения.
- ❷ Значение извлекается из карты конфигураций ConfigMap.
- ❸ Имя карты конфигураций ConfigMap.
- ❹ Ключ внутри карты конфигураций ConfigMap для поиска значения для переменной окружения.
- ❺ Значение извлекается из Secret (поиск выполняется так же, как при использовании карты конфигураций ConfigMap).

При таком подходе можно определять не только фактические значения переменных окружения (как для LOG_FILE), но и ссылаться на ресурсы Secret (предназначенные для хранения конфиденциальных данных) и ConfigMap (если настройки можно хранить в открытом виде). Преимущество использования ресурсов Secret и ConfigMap заключается в том, что они позволяют управлять переменными окружения независимо от определений подов. Подробнее о Secret и ConfigMap, а также об их плюсах и минусах рассказывается в главе 19 «Конфигурация в ресурсах».

В предыдущем примере значение для переменной SEED извлекается из ресурса Secret. В общем и целом это верное решение, но важно отметить, что переменные окружения не являются безопасными хранилищами. Конфиденциальная информация, помещенная в переменные окружения, становится доступной для чтения и может даже просочиться в журналы.

Пояснение

Переменные окружения просты в использовании и хорошо знакомы. Идея их применения хорошо отображается на контейнеры и поддерживается всеми платформами времени выполнения. Но переменные окружения небезопасны и хороши только для небольшого числа параметров. Но когда

параметров слишком много, управление переменными окружения превращается в утомительную задачу.

В таких случаях многие используют дополнительный уровень косвенности и помещают свои конфигурации в различные файлы, по одному для каждого окружения, а затем используют единственную переменную для выбора одного из этих файлов. Этот подход, например, используется в *профилях Spring Boot*. Поскольку файлы профилей обычно хранятся в самом приложении внутри контейнера, они оказываются тесно связанными с приложением. Это часто приводит к тому, что конфигурации для разработки и эксплуатации включаются в один образ Docker с приложением, что требует перестройки образа при любом изменении в любом окружении. Все это лишь доказывает, что переменные окружения подходят только для небольших наборов конфигураций.

Паттерны *Configuration Resource* (Конфигурация в ресурсах), *Immutable Configuration* (Неизменяемая конфигурация) и *Configuration Template* (Макет конфигурации), описанные в следующих главах, являются более удачными альтернативами, когда возникает потребность в более сложных конфигурациях.

Переменные окружения — это универсальный механизм, и их можно определять на разных уровнях. Это может приводить к фрагментированию настроек и затруднять выявление источника значения для заданной переменной. Из-за отсутствия централизованного места, где определяются все переменные окружения, часто бывает трудно отладить проблемы, связанные с ошибками в конфигурации.

Еще один недостаток переменных окружения состоит в том, что определить их можно только *перед* запуском приложения и нельзя изменить позже. С одной стороны, невозможность оперативного изменения конфигурации во время выполнения приложения можно считать недостатком. Однако многие видят

в этом преимущество, потому что это способствует *неизменности* конфигурации. В данном случае неизменность подразумевает необходимость остановки действующего контейнера и запуска новой копии с измененной конфигурацией, например, с помощью стратегии развертывания, такой как непрерывное развертывание обновлений. При таком подходе вы всегда будете иметь приложение с четко определенным состоянием конфигурации.

Переменные окружения просты в использовании, но в основном могут применяться только для хранения простых конфигураций и имеют ограничения при наличии более сложных требований. Паттерны, описываемые далее, показывают, как преодолеть эти ограничения.

Дополнительная информация

- Пример паттерна Конфигурация в переменных окружения (<http://bit.ly/2YcUtJC>).
- Методология «Двенадцать факторов» (<https://12factor.net/ru/>).
- Статья Кифа Морриса (Kief Morris) «Immutable Server» (<https://martinfowler.com/bliki/ImmutableServer.html>).
- Использование профилей Spring Boot для хранения конфигураций (<http://bit.ly/2YcSKUE>).

Глава 19. Конфигурация в ресурсах

Kubernetes предлагает свои ресурсы для хранения обычных и конфиденциальных конфигурационных данных, которые позволяют отделить жизненный цикл конфигурации от жизненного цикла приложения. Паттерн *Configuration Resource* (Конфигурация в ресурсах) объясняет идеи, заложенные в основу ресурсов ConfigMap и Secret, а также способы их использования и ограничения.

Задача

Один из существенных недостатков паттерна *EnvVar Configuration* (Конфигурация в переменных окружения) состоит в том, что он подходит для случаев с небольшим количеством параметров и простых конфигураций. Другой недостаток в том, что переменные окружения могут определяться в разных местах, из-за чего часто бывает сложно найти, где определяется та или иная переменная. И даже отыскав ее, нельзя быть полностью уверенным, что она не переопределяется где-то еще. Например, переменные, определяемые в образе Docker, можно переопределить во время выполнения в ресурсе Deployment.

Часто бывает удобнее хранить все конфигурационные данные в одном месте, а не во множестве файлов с определениями ресурсов. Также нет смысла помещать содержимое всего конфигурационного файла в переменную окружения. В таких случаях некоторая дополнительная косвенность может дать большую гибкость, что и предлагает паттерн *Configuration Resource* (Конфигурация в ресурсах).

Решение

Фреймворк Kubernetes предлагает специализированные ресурсы для хранения конфигураций, более гибкие, чем простые переменные окружения — объекты ConfigMap и Secret для данных общего назначения и конфиденциальных данных соответственно.

Оба ресурса поддерживают одинаковые способы использования, потому что обеспечивают хранение и управление парами ключ/значение. Все приемы, которые далее описываются применительно к ресурсам ConfigMap, в большинстве случаев можно использовать при работе с ресурсами Secret. Кроме кодирования в формат Base64, эти два ресурса не имеют технических различий.

После создания карты конфигурации ConfigMap и сохранения в ней данных есть два способа использования ключей из ConfigMap:

- как ссылки на *переменные окружения*, где ключ представляет имя переменной;
- как *файлы*, отображаемые в тома, смонтированные в поде. В этом случае ключ используется как имя файла.

Файл в смонтированном томе ConfigMap обновляется при изменении ConfigMap через Kubernetes API. То есть если приложение поддерживает горячую перезагрузку конфигурационных файлов, такое поведение позволяет получить определенные выгоды. Если ConfigMap используется для представления ссылок на переменные окружения, изменения в нем не отражаются на значениях переменных, потому что переменные окружения нельзя изменить после запуска процесса.

Другой альтернативой ресурсам ConfigMap и Secret является хранение конфигурации непосредственно на внешних томах, которые затем подключаются к поду.

Далее приводятся конкретные примеры использования ресурса ConfigMap, но они могут использоваться также с ресурсом Secret. Правда, есть одно важное обстоятельство: значения в ресурсе Secret хранятся в формате Base64.

Ресурс ConfigMap хранит пары ключ/значение в своем разделе data, как показано в листинге 19.1.

Листинг 19.1. Ресурс ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: random-generator-config
data:
  PATTERN: Configuration Resource ❶
  application.properties: |
    # Конфигурация генератора случайных чисел
    log.file=/tmp/generator.log
    server.port=7070
  EXTRA_OPTIONS: "high-secure,native"
  SEED: "432576345"
```

❶ Карты конфигураций ConfigMap могут быть доступны как переменные окружения и как смонтированные файлы. Мы советуем использовать символы верхнего регистра для обозначения ключей, которые используются как переменные окружения, и допустимые имена файлов при использовании монтируемых файлов.

Ресурс ConfigMap также может переносить содержимое полных конфигурационных файлов, как

`application.properties` для Spring Boot в этом примере. Нетрудно догадаться, что в нетривиальных случаях этот раздел может оказаться весьма большим!

Чтобы не создавать вручную полное описание ресурса, можно воспользоваться командой `kubectl`. Команда `kubectl`, которая воспроизводит предыдущий пример, показана в листинге 19.2.

Листинг 19.2. Создание ConfigMap из файла

```
kubectl create cm spring-boot-config \
    --from-literal=JAVA_OPTIONS=-
Djava.security.egd=file:/dev/urandom \
    --from-file=application.properties
```

Этот ресурс ConfigMap можно использовать везде, где доступны переменные окружения, как демонстрирует листинг 19.3.

Листинг 19.3. Установка переменной окружения из ConfigMap

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - env:
    - name: PATTERN
      valueFrom:
        configMapKeyRef:
          name: random-generator-config
          key: PATTERN
  . . . .
```

Если в ConfigMap присутствует большое число записей, которые должны быть доступны в виде переменных окружения, использование специального синтаксиса может сэкономить немало времени. Вместо объявления каждой записи по отдельности, как показано в предыдущем примере в разделе `env:`, можно определить раздел `envFrom:`, который отобразит все записи из ConfigMap с ключами, которые можно использовать как допустимые имена переменных окружения. Также есть возможность добавить определенный префикс, как показано в листинге 19.4.

Листинг 19.4. Установка переменных окружения из значений в ConfigMap

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
    envFrom: ❶
    - configMapRef:
      name: random-generator-config
      prefix: CONFIG_ ❷
```

❶ Выбрать все ключи из ресурса ConfigMap с именем `random-generator-config`, которые можно использовать в качестве имен переменных окружения.

❷ Все отобранные ключи из ConfigMap предваряются префиксом `CONFIG_`. В результате для ConfigMap из примера 19.1 будут созданы переменные `CONFIG_PATTERN_NAME`, `CONFIG_EXTRA_OPTIONS` и `CONFIG_SEED`.

Ресурсы Secret тоже можно использовать для создания переменных окружения, как и ConfigMap, либо из каждой записи в отдельности, либо сразу из всех записей. Чтобы

задействовать `Secret` вместо `ConfigMap`, нужно заменить `configMapKeyRef` на `secretKeyRef`.

При использовании в качестве тома весь ресурс `ConfigMap` проецируется в указанный том, при этом ключи используются как имена файлов (см. листинг 19.5).

Листинг 19.5. Монтирование `ConfigMap` как тома

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    volumeMounts:
    - name: config-volume
      mountPath: /config
  volumes:
  - name: config-volume
    configMap: ❶
    name: random-generator-config
```

❶ Том на основе `ConfigMap` будет содержать файлы по числу записей в ресурсе. Ключи будут использованы как имена файлов, а значения — как их содержимое.

Конфигурация из листинга 19.1, смонтированная как том, создаст два файла в папке `/config`: `application.properties` с содержимым, определенным в `ConfigMap`, и файл `PATTERN` с единственной строкой.

Добавляя дополнительные свойства в объявление тома, можно более тонко настроить отображение конфигурационных

данных. Вместо отобра-

насколько защищены ресурсы Secret?

Ресурсы Secret хранят данные в формате Base64 и декодируют их перед передачей в переменные окружения или смонтированные тома. Многие ошибочно считают такое кодирование безопасным. Base64 не является алгоритмом шифрования и с точки зрения безопасности данные в этом формате являются не более защищенными, чем открытый текст. Формат Base64 в Secret просто позволяет хранить двоичные данные. Тогда почему ресурсы Secret считаются более безопасными, чем ресурсы ConfigMap? Потому что есть еще ряд деталей реализации Secret, которые делают эти ресурсы безопасными. В этом направлении постоянно появляются новые улучшения, но к основным деталям реализации относятся следующие:

- ресурсы Secret передаются только на узлы, где действуют поды, использующие их;
- на узлах ресурсы Secret хранятся в памяти, в файловой системе `tmpfs`, никогда не записываются в физическое хранилище и удаляются после остановки пода;
- в Etcd ресурсы Secret хранятся в зашифрованном виде.

Как бы то ни было, есть вполне законные способы получить доступ к ресурсам Secret, например, обратившись от имени пользователя `root` или даже создав под и смонтировав нужный

ресурс Secret в него. Можно, конечно, применить приемы управления доступом на основе ролей (Role-Based Access Control, RBAC) к ресурсам Secret (так же, как для ConfigMap или других ресурсов) и разрешить их чтение только определенным подам с предопределенными учетными записями служб. Пользователи, имеющие возможность создавать поды в пространстве имен, могут повышать свои привилегии в этом пространстве имен, запускать поды под учетной записью с более высокими привилегиями и читать ресурсы Secret. Пользователь или контроллер с привилегиями, разрешающими создавать поды в пространстве имен, сможет действовать от имени любой учетной записи службы и получить доступ ко всем ресурсам Secret и ConfigMap в этом пространстве имен. По этой причине на уровне приложений часто выполняется дополнительное шифрование конфиденциальной информации.

жения всех записей в файлы можно выбрать конкретные ключи и имена файлов, под которыми они должны быть доступны. За более подробной информацией обращайтесь к документации с описанием ConfigMap.

Еще один способ хранения конфигурации в Kubernetes — использование томов gitRepo. Тома этого типа монтируются в пустые каталоги в подах, и в них клонируются заданные репозитории Git. Хранение конфигурации в Git автоматически дает возможность управления версиями и аудита. Но тома gitRepo требуют доступа к внешним репозиториям Git, которые не являются ресурсами Kubernetes, могут находиться вне кластера и требовать отдельного мониторинга и управления. Клонирование и монтирование происходит во время запуска пода, и локальная копия не обновляется

автоматически при изменении основного репозитория. Тома этого типа работают подобно паттерну *Immutable Configuration* (Неизменяемая конфигурация), описанному в главе 20, и с использованием паттерна *Init Container* (Init-контейнер) для копирования конфигурации в локальный том.

В настоящее время тома типа `gitRepo` считаются устаревшими и вместо них рекомендуется использовать решения на основе паттерна *Init Container* (Init-контейнер), потому что эти решения более универсальны и поддерживают другие источники данных, а не только Git. Вы тоже можете использовать этот подход для извлечения конфигурационных данных из внешних систем и сохранения их в томе, но вместо предопределенного тома `gitRepo` использовать более гибкий метод *Init Container* (Init-контейнер). Подробнее этот прием мы обсудим в разделе «Init-контейнеры в Kubernetes» главы 20.

Пояснение

Ресурсы `ConfigMap` и `Secret` позволяют хранить конфигурационную информацию в специализированных объектах ресурсов, которые легко управляются с помощью Kubernetes API. Самое большое преимущество `ConfigMap` и `Secret` в том, что они отделяют *определение* конфигурационных данных от их *использования*. Такое разделение позволяет управлять объектами, используя конфигурации независимо от конфигураций.

Еще одно преимущество в том, что `ConfigMap` и `Secret` являются неотъемлемой частью платформы. Они не требуют никаких нестандартных конструкций, подобных тем, что описаны в главе 20 «Неизменяемая конфигурация».

Тем не менее паттерн *Configuration Resource* (Конфигурация в ресурсах) тоже имеет свои ограничения: ресурсы `Secret` не

могут иметь размер больше 1 Мбайт, то есть они не могут хранить произвольные объемы данных и не годятся для хранения прикладных данных, не имеющих отношения к конфигурации. Ресурсы Secret могут хранить двоичные данные, но из-за того что данные хранятся в формате Base64, чистый их объем не может превышать 700 Кбайт.

Кроме того, многие действующие кластеры Kubernetes устанавливают свои квоты на число ресурсов ConfigMap, которые можно использовать в пространстве имен или в проекте, поэтому ConfigMap тоже не является универсальным решением.

В следующих двух главах мы познакомимся с приемами хранения очень больших объемов конфигурационных данных, основанных на паттернах *Immutable Configuration* (Неизменяемая конфигурация) и *Configuration Template* (Макет конфигурации).

Дополнительная информация

- Пример конфигурации в ресурсах (<http://bit.ly/2YeGymi>).
- Документация с описанием ConfigMap (<http://bit.ly/2Cs59uQ>).
- Документация с описанием Secret (<https://kubernetes.io/docs/concepts/configuration/secret/>).
- Стационарное шифрование данных в Secret (<http://bit.ly/2ORsavn>).
- Безопасное распространение учетных данных с помощью Secret (<http://bit.ly/2FfcvCn>).
- Тома gitRepo (<http://bit.ly/2HxuGqO>).

- Ограничения на размеры ConfigMap (<http://bit.ly/2UkHRRy>).

Глава 20. Неизменяемая конфигурация

Паттерн *Immutable Configuration* (Неизменяемая конфигурация) предполагает упаковку конфигурационных данных в неизменяемый образ контейнера и его подключение к приложению во время выполнения. Это не только дает возможность использовать разные и неизменяемые версии конфигурационных данных, но и позволяет преодолеть ограничения на объем этих данных, свойственные переменным окружения и ресурсам ConfigMap.

Задача

Как рассказывалось в главе 18 «Конфигурация в переменных окружения», переменные окружения предлагают самый простой способ настройки контейнерных приложений. Но несмотря на простоту и универсальность, управлять переменными окружения становится крайне сложно, как только их число превысит определенный порог.

Эту сложность отчасти можно преодолеть с помощью *Configuration Resources* (Конфигурация в ресурсах). Однако эти паттерны не обеспечивают *неизменности* самих конфигурационных данных. Под неизменностью здесь подразумевается невозможность изменить конфигурацию после запуска приложения, которая гарантирует, что конфигурационные данные всегда находятся в четко определенном состоянии. Кроме того, неизменяемую конфигурацию можно хранить в системе управления версиями и следовать за процессом контроля изменений.

Решение

Для решения упомянутых проблем все конфигурационные данные для конкретного окружения можно поместить в один пассивный образ и распространять его как обычный образ контейнера. Во время выполнения приложение и образ с данными связываются друг с другом, и приложение получает возможность извлекать конфигурацию из этого образа. Такой подход позволяет легко создавать различные образы с конфигурационными данными для различных окружений. Эти образы объединяют всю информацию о конфигурации для конкретных окружений и могут храниться в системе управления версиями, как и любые другие образы контейнеров.

Создаются такие образы данных легко и просто, так как являются обычными образами контейнеров, которые хранят только данные. Сложность заключается лишь в том, чтобы связать контейнеры во время запуска. Для этого можно использовать различные подходы, в зависимости от платформы.

Тома Docker

Прежде чем рассматривать решение для Kubernetes, отступим на шаг назад и рассмотрим случай с обычными контейнерами Docker. В Docker контейнер может экспортировать *том* с данными. С помощью директивы `VOLUME` в *Dockerfile* можно указать каталог, который позже можно сделать общим. Во время запуска содержимое этого каталога в контейнере копируется в общий каталог. Как показано на рис. 20.1, такое связывание томов

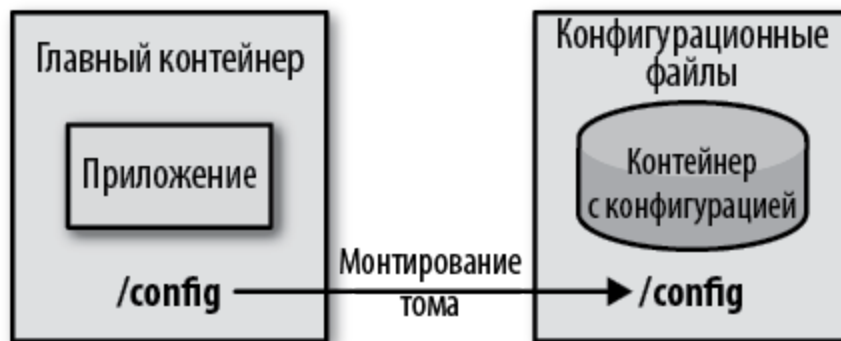


Рис. 20.1. Реализация паттерна Immutable configuration (Неизменяемая конфигурация) с использованием тома Docker

является отличным способом передачи конфигурационной информации из контейнера с конфигурацией в контейнер приложения.

Рассмотрим пример. Создадим для окружения разработки образ Docker, который содержит ее конфигурацию и создает том `/config`. Создать такой образ можно с помощью `Dockerfile-config`, как показано в листинге 20.1.

Листинг 20.1. `Dockerfile`, определяющий образ с конфигурацией

```
FROM scratch

ADD                                app-dev.properties
/config/app.properties ❶

VOLUME                             ❷
/config
```

❶ Добавить заданное свойство.

❷ Создать том и скопировать свойство в него.

Теперь, с помощью клиента командной строки `Docker CLI`, создадим сам образ и контейнер `Docker`, как показано в листинге 20.2.

Листинг 20.2. Сборка образа `Docker` с конфигурацией

```
docker build -t k8spatterns/config-dev-  
image:1.0.1 -f  
        Dockerfile-config  
docker create --name config-dev  
k8spatterns/config-dev-image:1.0.1 .
```

Заключительный шаг: запуск контейнера с приложением и связывание его с конфигурационным контейнером (листинг 20.3).

Листинг 20.3. Запуск контейнера с приложением и связывание его с конфигурационным контейнером

```
docker run --volumes-from config-dev  
k8spatterns/welcome-servlet:1.0
```

Образ с приложением ожидает найти файлы конфигурации в каталоге */config*, в томе, смонтированном контейнером с конфигурацией. После перемещения приложения из окружения разработки в промышленное окружение вам останется только изменить команду запуска. Менять сам образ приложения не придется, вы просто свяжете контейнер приложения с контейнером, содержащим конфигурацию промышленного окружения, как показано в листинге 20.4.

Листинг 20.4. Использование другой конфигурации в промышленном окружении

```
docker build -t k8spatterns/config-prod-  
image:1.0.1 -f  
        Dockerfile-config  
docker create --name config-prod k8spatterns/  
        config-prod-image:1.0.1 .  
docker run --volumes-from config-prod  
k8spatterns/  
        welcome-servlet:1.0
```

Init-контейнеры в Kubernetes

Совместное использование томов контейнерами в поде идеально подошло бы для связывания контейнеров приложений с конфигурационными контейнерами в Kubernetes. Но если мы решим перенести эту технику из Docker в мир Kubernetes, то обнаружим, что в настоящее время в Kubernetes нет поддержки контейнерных томов. Учитывая, насколько давно обсуждается необходимость этой поддержки, сложность ее реализации и ограниченность преимуществ, представляется маловероятным, что контейнерные тома появятся в ближайшее время.

Поэтому контейнеры могут совместно использовать внешние тома, но пока не могут напрямую использовать общие каталоги, находящиеся в самих контейнерах. Однако для реализации паттерна *Immutable Configuration* (Неизменяемая конфигурация) в Kubernetes можно использовать паттерн *Init Containers* (Init-контейнеры), описанный в главе 14, который может инициализировать пустой общий том во время запуска.

В примере с контейнерами Docker мы создали пустой образ с конфигурацией, не содержащий никаких файлов операционной системы. Нам ничего не нужно иметь в этом контейнере, кроме конфигурационных данных, передаваемых через тома Docker. Однако init-контейнеру в Kubernetes нужна некоторая помощь от базового образа, чтобы скопировать конфигурационные данные в общий том в поде. `busybox` — хороший выбор на роль такого базового образа, потому что он достаточно мал и позволяет использовать простую Unix-команду `cp`.

Но как происходит инициализация общих томов с конфигурациями? Рассмотрим пример. Для начала снова

создадим образ с конфигурацией с помощью *Dockerfile*, как показано в листинге 20.5.

Листинг 20.5. Образ с конфигурацией для окружения разработки

```
FROM busybox

ADD dev.properties /config-src/demo.properties

ENTRYPOINT [ "sh", "-c", "cp /config-src/* $1",
"--" ] ❶
```

❶ Здесь для интерпретации шаблонных символов используется командная оболочка.

Единственное отличие от примера 20.1 со стандартным образом Docker заключается в наличии базового образа и директивы `ENTRYPOINT` с кодом, который копирует файл свойств в каталог, указанный в аргументе команды запуска образа Docker. Теперь на этот образ можно сослаться в `init`-контейнере, внутри раздела `.template.spec` с определением развертывания `Deployment` (листинг 20.6).

Листинг 20.6. Определение развертывания, копирующего конфигурацию из `init`-контейнера в указанный каталог

```
initContainers:
- image: k8spatterns/config-dev:1
  name: init
  args:
  - "/config"
  volumeMounts:
  - mountPath: "/config"
    name: config-directory
containers:
- image: k8spatterns/demo:1
  name: demo
```

```
ports:
- containerPort: 8080
  name: http
  protocol: TCP
volumeMounts:
- mountPath: "/config"
  name: config-directory
volumes:
- name: config-directory
  emptyDir: {}
```

Определение развертывания Deployment включает один том и два контейнера:

- Том `volume config-directory` имеет тип `emptyDir`, то есть на узле, где находится под, создается пустой каталог.
- Init-контейнер, вызываемый фреймворком Kubernetes во время запуска, конструируется из образа, который мы только что создали. Мы также определили единственный аргумент `/config`, который передается в `ENTRYPOINT` образа. Этот аргумент сообщает init-контейнеру, куда следует скопировать его содержимое. Каталог `/config` монтируется из тома `config-directory`.
- Контейнер приложения монтирует том `config-directory`, чтобы получить доступ к конфигурации, скопированной из init-контейнера.

Схема на рис. 20.2 иллюстрирует, как контейнер приложения получает доступ к конфигурационным данным, созданным init-контейнером в общем томе.

Теперь, чтобы заменить конфигурацию для окружения разработки конфигурацией для промышленного окружения, нужно лишь заменить образ init-контейнера. Для этого можно исправить определение YAML или произвести обновление с помощью команды `kubectl`. Однако было бы слишком утомительно править описание ресурса для каждого окружения. Если вы используете Red Hat OpenShift, корпоративный дистрибутив Kubernetes, *паттерны OpenShift* могут помочь в решении этой проблемы. Паттерны OpenShift дают возможность создавать разные описания ресурсов для разных окружений из одного паттерна.

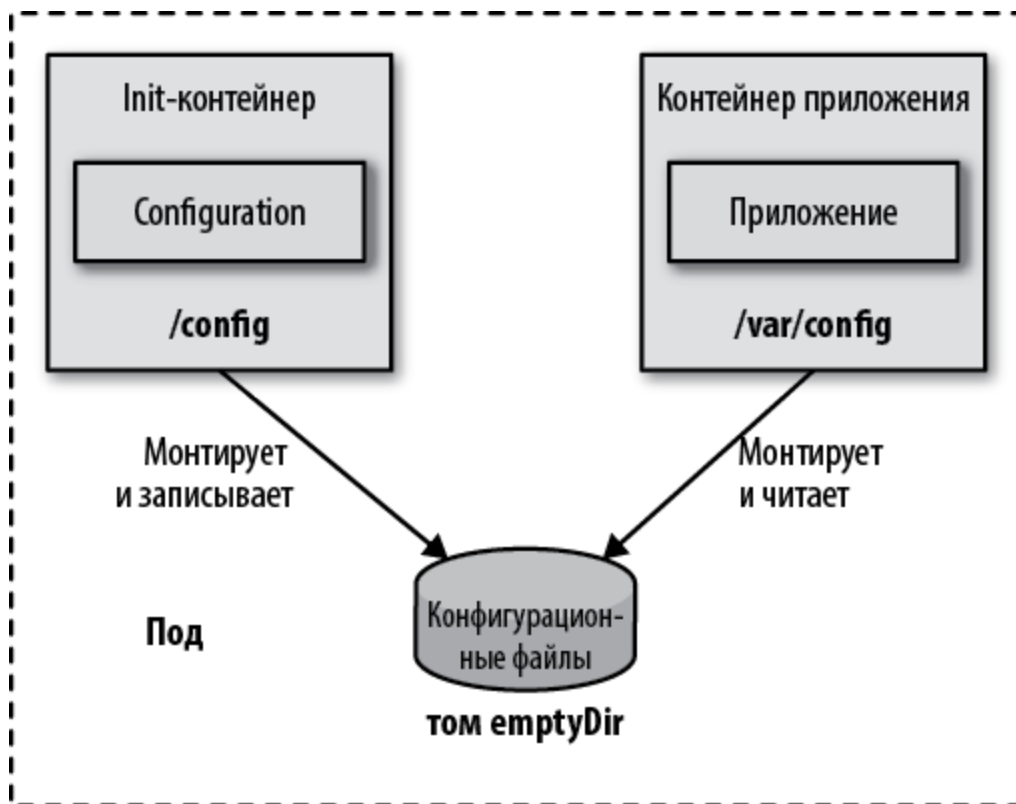


Рис. 20.2. Реализация паттерна Immutable Configuration (Неизменяемая конфигурация) с помощью init-контейнера

Паттерны OpenShift

Паттерны — это обычные описания ресурсов, которые допускают параметризацию. Как показано в листинге 20.7, в качестве такого параметра можно использовать образ конфигурации.

Листинг 20.7. Паттерн OpenShift, параметризуемый образом с конфигурацией

```
apiVersion: v1
kind: Template
metadata:
  name: demo
parameters:
  - name: CONFIG_IMAGE ❶
    description: Name of configuration image
    value: k8spatterns/config-dev:1
objects:
- apiVersion: v1
  kind: DeploymentConfig
  // ....
  spec:
    template:
      metadata:
        // ....
      spec:
        initContainers:
          - name: init
            image: ${CONFIG_IMAGE} ❷
            args: [ "/config" ]
            volumeMounts:
              - mountPath: /config
                name: config-directory
        containers:
          - image: k8spatterns/demo:1
```

```
// ...
volumeMounts:
  - mountPath: /config
    name: config-directory
volumes:
  - name: config-directory
    emptyDir: {}
```

- ❶ Объявление параметра паттерна CONFIG_IMAGE.
- ❷ Использование параметра паттерна.

Здесь показана только часть полного описания, но мы с легкостью сможем распознать параметр CONFIG_IMAGE в объявлении init-контейнера. Если создать этот паттерн в кластере OpenShift, мы сможем создать его экземпляр командой `oc`, как показано в листинге 20.8.

Листинг 20.8. Применение паттерна OpenShift для создания нового приложения

```
oc          new-app          demo          -p
CONFIG_IMAGE=k8spatterns/config-prod:1
```

Как обычно, инструкции по опробованию этого примера, а также полное определение контроллера Deployment можно найти в репозитории Git с примерами для этой книги.

Пояснение

Использование отдельных контейнеров с данными в паттерне *Immutable Configuration* (Неизменяемая конфигурация) — не самая простая задача. Однако этот паттерн предлагает некоторые уникальные преимущества:

- Конфигурация для конкретного окружения заключена в контейнер, а значит, ее можно хранить в системе

управления версиями, как любые другие образы контейнеров.

- Конфигурацию, созданную таким способом, можно распространять через реестр контейнеров. Кроме того, такую конфигурацию можно проверить, даже не имея доступа к кластеру.
- Конфигурация считается неизменяемой, потому что хранится в образе контейнера: изменение конфигурации требует обновления версии и создания нового образа контейнера.
- Образы с конфигурационными данными удобнее, когда конфигурации слишком сложны или содержат слишком большие объемы конфигурационных данных, чтобы их можно было поместить в переменные окружения или в ресурсы ConfigMap.

Однако этот паттерн имеет и свои недостатки:

- Он сложнее в использовании, потому что приходится создавать и распространять через реестры дополнительные образы контейнеров.
- Он не решает проблемы безопасности конфиденциальных конфигурационных данных.
- В случае с Kubernetes требуется дополнительная обработка `init`-контейнеров, а значит, придется управлять различными объектами Deployment для разных окружений.

В любом случае нужно тщательно оценить, действительно ли необходим такой сложный подход. Если неизменность не

требуется, может быть, будет достаточно простого ресурса ConfigMap, как описано в главе 19 «Конфигурация в ресурсах».

Еще одно решение, когда необходимо работать с большими конфигурационными файлами, незначительно отличающимися в разных окружениях, описано в следующей главе, где обсуждается паттерн *Configuration Template* (Макет конфигурации).

Дополнительная информация

- Пример неизменяемой конфигурации (<http://bit.ly/2HL95dp>).
- Как имитировать `--volumes-from` в Kubernetes (<http://bit.ly/2YbRhhy>).
- Предложение по улучшению: поддержка томов образов в Kubernetes (<http://bit.ly/2Wf0pjt>).
- `docker-flexvol`: драйвер для Kubernetes, поддерживающий тома Docker (<https://github.com/dims/docker-flexvol>).
- Паттерны OpenShift (<https://red.ht/2Ohh7v0>).

Глава 21. Макет конфигурации

Паттерн *Configuration Template* (Макет конфигурации) позволяет создавать и обрабатывать большие и сложные конфигурации на этапе запуска приложения. Сгенерированная конфигурация зависит от целевого окружения времени выполнения, определяемого параметрами, которые используются при обработке макета конфигурации.

Задача

В главе 19 «Конфигурация в ресурсах» вы узнали, как использовать стандартные объекты ресурсов Kubernetes — ConfigMap и Secret — для настройки приложений. Но иногда конфигурационные файлы могут быть очень большими и сложными. Поместить такие файлы непосредственно в ConfigMaps может быть проблематично, поскольку они должны быть правильно внедрены в определения ресурсов. Также приходится быть очень осторожными и избегать специальных символов, таких как кавычки и переносы строк, играющих особую роль в синтаксисе определения ресурсов Kubernetes. Размер конфигурации — еще один важный фактор, который следует учитывать, потому что ресурсы ConfigMap или Secret ограничивают суммарный объем всех значений величиной 1 Мбайт (это ограничение накладывается внутренним хранилищем Etcd).

Большие конфигурационные файлы для разных окружений обычно имеют незначительные отличия. Такое их сходство приводит к большому количеству повторений и избыточности в ресурсах ConfigMap, потому что в каждом окружении используются в основном одни и те же данные. Паттерн

Configuration Template (Макет конфигурации), который мы рассмотрим в этой главе, решает эти конкретные проблемы.

Решение

Чтобы уменьшить объем повторяющихся данных, имеет смысл хранить в ConfigMap или даже непосредственно в переменных окружения только различающиеся конфигурационные значения, такие как параметры подключения к базе данных. Во время запуска контейнера эти значения обрабатываются с помощью паттерна *Configuration Template* (Макет конфигурации), чтобы получить полный конфигурационный файл (как, например, *standalone.xml* в JBoss WildFly). Существует большое число инструментов, таких как Tiller (Ruby) или Gomplate (Go), для обработки макетов на этапе инициализации приложения. На рис. 21.1 показан пример макета конфигурации, заполненный данными, поступающими из переменных окружения или смонтированного тома, возможно, основанного на ConfigMap.

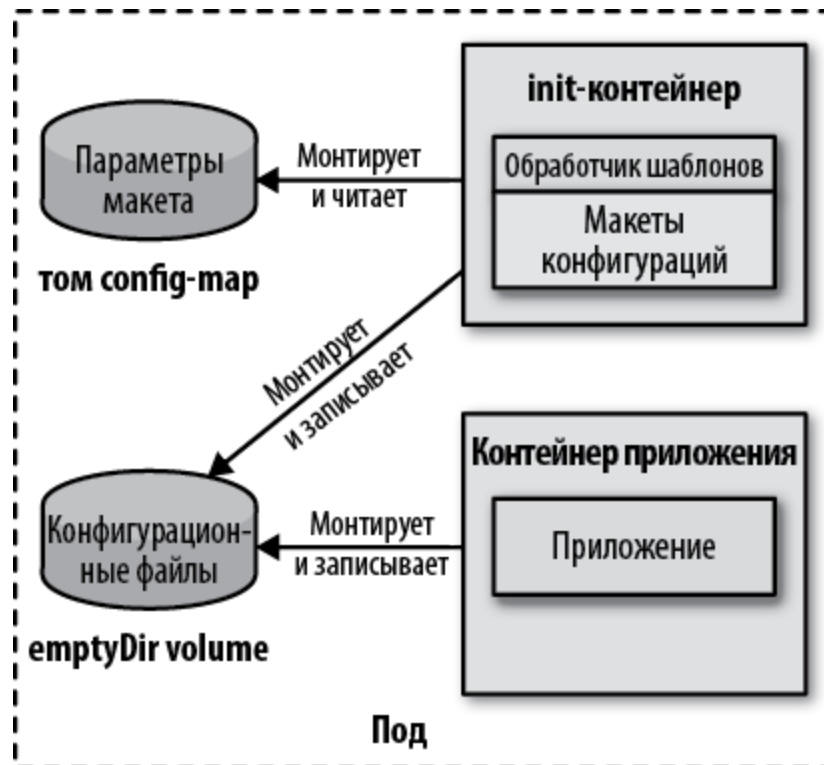


Рис. 21.1. Макет конфигурации

Перед запуском приложения подготовленный конфигурационный файл помещается туда, откуда его можно напрямую использовать, как любой другой файл конфигурации.

Вот два способа, как можно реализовать такую оперативную обработку во время запуска:

- Можно добавить обработчик паттернов как часть ENTRYPOINT в *Dockerfile*, чтобы сделать обработку макета непосредственной частью образа контейнера. Роль точки входа в таких случаях обычно играет сценарий, который сначала обрабатывает макет, а потом запускает приложение. Параметры для макета извлекаются из переменных окружения.

- Лучший способ выполнить инициализацию в Kubernetes — использовать `init`-контейнер с обработчиком макетов, который создает конфигурацию для контейнеров приложений в поде. `Init`-контейнеры подробно описываются в главе 14.

Решение с использованием `init`-контейнеров выглядит более привлекательным в Kubernetes, потому что позволяет использовать `ConfigMap` для хранения параметров макета. Схема на рис. 21.1 иллюстрирует его работу.

Определение пода приложения включает как минимум два контейнера: `init`-контейнер для обработки макета и контейнер приложения. `Init`-контейнер содержит не только обработчик макетов, но и сами макеты конфигурации. Кроме контейнеров, этот под определяет также два тома: один для параметров паттерна, основанный на `ConfigMap`, и том `emptyDir`, используемый для совместного использования обработанных макетов `init`-контейнером и контейнером приложения.

При такой организации во время запуска этого пода выполняются следующие шаги:

1. `Init`-контейнер запускается и вызывает обработчик макетов. Обработчик извлекает макеты из своего образа и параметры из смонтированного тома `ConfigMap` и сохраняет результат в томе `emptyDir`.
2. После завершения `init`-контейнера запускается контейнер приложения и загружает конфигурационные файлы из тома `emptyDir`.

Следующий пример использует `init`-контейнер для управления полным набором конфигурационных файлов

WildFly для двух окружений: окружения разработки и окружения промышленной эксплуатации. Оба очень похожи друг на друга и имеют мало отличий. На самом деле в нашем примере они отличаются только способом журналирования: каждая строка в журнале снабжается префиксом DEVELOPMENT: или PRODUCTION: соответственно.

Полный пример с исчерпывающими инструкциями по установке можно найти в репозитории GitHub (<http://bit.ly/2TKUNZY>), а здесь будет показана только основная идея.

Макет конфигурации журналирования в листинге 21.1 хранится в файле *standalone.xml* и параметризуется с использованием синтаксиса паттернов Go.

Листинг 21.1. Макет конфигурации журналирования

```
.....
<formatter name="COLOR-PATTERN">
    <pattern-formatter pattern="{{(datasource
"config").logFormat}}"/>
</formatter>
.....
```

Здесь для обработки макетов применяется процессор паттернов Gomplate, который использует понятие *источника данных* для ссылки на параметры макета, которые требуется заполнить. В нашем случае источником данных является том на основе ConfigMap, смонтированный в init-контейнер. В данном случае ConfigMap содержит одну запись с ключом logFormat, из которой извлекается фактический формат.

Теперь на основе этого макета можно создать Docker-образ init-контейнера. *Dockerfile* для образа *k8spatterns/example-configuration-template-init* очень прост (листинг 21.2).

Листинг 21.2. Простой Dockerfile, определяющий образ макета

```
FROM k8spatterns/gomplate
COPY in /in
```

Базовый образ *k8spatterns/gomplate* определяет обработчик макетов и сценарий точки входа, который по умолчанию использует следующие каталоги:

- */in* содержит макеты конфигурации для WildFly, включая параметризованный *standalone.xml*. Они добавляются непосредственно в образ.
- */params* используется для поиска источников данных Gomplate — файлов YAML. Этот каталог монтируется из тома ConfigMap.
- */out* — это каталог, куда сохраняются обработанные файлы. Этот каталог монтируется в контейнер приложения WildFly и используется для конфигурации.

Второй ингредиент нашего примера — ресурс ConfigMap, определяющий параметры. В листинге 21.3 мы используем простой файл с парами ключ/значение.

Листинг 21.3. Значения для макета конфигурации

```
logFormat: "DEVELOPMENT: %-5p %s%e%n"
```

Ресурс ConfigMap с именем *wildfly-parameters* определяет эти данные в формате YAML с ключом *config.yml*, который используется *init*-контейнером для извлечения данных.

Наконец, нам нужно определить ресурс Deployment для развертывания сервера WildFly (листинг 21.4).

Листинг 21.4. Развертывание с использованием *init*-контейнера в роли обработчика макета

```
apiVersion: extensions/v1beta1
```

```

kind: Deployment
metadata:
  labels:
    example: cm-template
  name: wildfly-cm-template
spec:
  replicas: 1
  template:
    metadata:
      labels:
        example: cm-template
    spec:
      initContainers:
        - image: k8spatterns/example-config-cm-
          template-init ❶
          name: init
          volumeMounts:
            - mountPath:
              "/params" ❷
              name: wildfly-parameters
            - mountPath:
              "/out" ❸
              name: wildfly-config
      containers:
        - image: jboss/wildfly:10.1.0.Final
          name: server
          command:
            -
              "/opt/jboss/wildfly/bin/standalone.sh"
            - "-Djboss.server.config.dir=/config"
          ports:

```

```

- containerPort: 8080
  name: http
  protocol: TCP
  volumeMounts:
    - mountPath:
      ④
      "/config"
      name: wildfly-config
    volumes:
      ⑤
      - name: wildfly-parameters
        configMap:
          name: wildfly-parameters
      - name: wildfly-config
        emptyDir: {}

```

① Образ с макетами конфигураций.

② Параметры из ConfigMap `wildfly-parameters`.

③ Целевой каталог для записи обработанных макетов. Монтируется из пустого тома.

④ Каталог для записи сгенерированных конфигурационных файлов, монтируемый как `/config`.

⑤ Объявления тома ConfigMap с параметрами и пустого каталога, используемого для передачи обработанной конфигурации.

Это довольно длинное объявление, поэтому рассмотрим его подробнее: определение Deployment содержит описание пода с нашим `init`-контейнером, контейнером приложения и двумя внутренними томами:

- Первый том, `wildfly-parameters`, содержит наш ресурс ConfigMap с тем же именем (то есть он содержит файл `config.yml` со значением параметра).

- Второй том изначально является пустым каталогом и используется совместно `init`-контейнером и контейнером `WildFly`.

Если запустить это развертывание `Deployment`, произойдет следующее:

- Будет создан `init`-контейнер и выполнится определяемая им команда. Этот контейнер извлечет файл `config.yml` из тома `ConfigMap`, заполнит макеты из каталога `/in` в `init`-контейнере и сохранит получившиеся файлы в каталоге `/out`. Каталог `/out` — это точка монтирования тома `wildfly-config`.
- После того как `init`-контейнер завершится, будет запущен сервер `WildFly` с параметром, указывающим, что полная конфигурация находится в каталоге `/config`. И снова, `/config` — это общий том `wildfly-config`, содержащий обработанные файлы макета.

Важно отметить, что это описание ресурса развертывания `Deployment` *не* придется изменять при переносе приложения из окружения разработки в окружение промышленной эксплуатации. Изменить понадобится только `ConfigMap` с параметрами макета.

С помощью этого метода легко создать конфигурацию `DRY`¹⁷ без копирования и обслуживания повторяющихся больших конфигурационных файлов. Например, если понадобится внести одинаковые изменения в конфигурации `WildFly` для всех окружений, достаточно будет изменить только один файл макета в `init`-контейнере. Это дает значительные преимущества для тех, кто занимается обслуживанием, потому

что устраняет опасность несогласованного изменения конфигураций.



Советы по отладке томов

При работе с подами и томами, как в этом паттерне, не совсем понятно, как отлаживать проблемы, если паттерн работает не так, как ожидалось. Если понадобится проверить обработанные макеты, проверьте каталог `/var/lib/kubelet/pods/ndompodid/volumes/kubernetes.io~empty-dir/` на узле, потому что именно там находится содержимое тома `emptyDir`. Просто выполните `kubectl exec` в поде после его запуска и проверьте этот каталог на наличие созданных файлов.

Пояснение

Паттерн *Configuration Template* (Макет конфигурации) построен на основе паттерна *Configuration Resource* (Конфигурация в ресурсах) и прекрасно подходит для ситуаций, когда приложение требуется запускать в разных окружениях со сложными и похожими конфигурациями. Однако настройка приложений с использованием паттерна *Configuration Template* (Макет конфигурации) более сложна и включает больше компонентов, которые могут работать неправильно. Используйте его, только если приложение требует огромного

объема конфигурационных данных. Часто такие конфигурационные данные имеют весьма небольшие отличия в разных окружениях. Даже если копирование всей конфигурации для конкретного окружения непосредственно в ConfigMap позволяет добиться желаемого, такой подход усложнит обслуживание конфигураций в будущем, потому что со временем конфигурации неизбежно будут расходиться все дальше друг от друга. Для таких случаев паттерн *Configuration Template* (Макет конфигурации) подходит идеально.

Дополнительная информация

- Пример реализации макета конфигурации (<http://bit.ly/2TKUHZY>).
- Механизм обработки макетов Tiller (<https://github.com/markround/tiller>).
- Gomplate (<https://github.com/hairyhenderson/gomplate>).
- Синтаксис макетов Go (<https://golang.org/pkg/html/template/>).

¹⁷ DRY — аббревиатура от англ. «Don't Repeat Yourself» — не повторяйся.

Часть V. Дополнительные паттерны

Паттерны, относящиеся к этой категории, охватывают более сложные ситуации, которые не вписываются ни в одну из предыдущих категорий. Некоторые из паттернов, такие как *Controller* (Контроллер), появились очень давно и на них построен сам фреймворк Kubernetes. Но есть и другие, считающиеся очень новыми (такие, как Knative, для создания образов контейнеров и масштабирования служб Service до нуля), которые могут измениться к тому моменту, когда вы будете читать эти строки. Чтобы не отстать, мы будем постоянно обновлять наши онлайн-примеры (<https://github.com/k8spatterns/examples>) и отражать последние достижения в этой области.

Вот эти дополнительные паттерны, которые мы рассмотрим в следующих главах:

- Глава 22 «Контроллер» описывает паттерн *Controller* (Контроллер), имеющий большое значение для самого фреймворка Kubernetes. Этот паттерн демонстрирует, как можно расширить возможности платформы с помощью своих контроллеров.
- Глава 23 «Оператор» описывает способ использования паттерна *Controller* (Контроллер) в сочетании с собственными и предметно-ориентированными ресурсами для представления практических знаний в автоматизированной форме.
- Глава 24 «Эластичное масштабирование» рассказывает, как Kubernetes справляется с динамическими нагрузками путем масштабирования в разных направлениях.

- Глава 25 «Построитель образов» описывает паттерн, который переносит задачу сборки образов в сам кластер.

Глава 22. Контроллер

Контроллер ведет активный мониторинг и поддерживает набор ресурсов Kubernetes в нужном состоянии. Основа самого фреймворка Kubernetes состоит из парка контроллеров, которые регулярно проверяют и согласовывают текущее состояние приложений с требуемым целевым состоянием. В этой главе вы увидите, как использовать эту базовую идею для расширения платформы под свои нужды.

Задача

Вы уже знаете, что Kubernetes — это сложная и многогранная платформа, предлагающая множество возможностей. Однако это универсальная платформа управления контейнерами охватывает далеко не все варианты использования приложений. К счастью, она поддерживает точки расширения, используя которые можно реализовать конкретные варианты, опираясь на проверенные строительные блоки Kubernetes.

Главный вопрос, который мы рассмотрим здесь: как расширить Kubernetes, не изменяя и не нарушая его работу, и как использовать его возможности для поддержки нестандартных сценариев.

В основе Kubernetes лежит декларативный API, ориентированный на ресурсы. Что подразумевается под словом *декларативный*? Декларативный подход, в противоположность *императивному*, описывает не как должен действовать Kubernetes, а каким должно быть целевое состояние. Например, масштабируя развертывание Deployment вверх, мы не создаем новые поды, требуя от Kubernetes «создать новый

под», а меняем свойство `replicas` ресурса `Deployment` через `Kubernetes API`, присваивая ему желаемое число.

Но как создаются новые поды? Эту задачу решают внутренние контроллеры. При каждом изменении состояния ресурса (например, при изменении значения свойства `replicas` в развертывании `Deployment`) `Kubernetes` создает событие и передает его всем обработчикам. Затем эти обработчики реагируют на событие, изменяя, удаляя или создавая новые ресурсы, что, в свою очередь, приводит к появлению других событий, таких как событие, требующее создать новый под. Эти события затем передаются другим контроллерам, которые выполняют свои конкретные действия.

Весь этот процесс известен как *согласование состояний*, когда целевое состояние (требуемое количество реплик) отличается от текущего (фактическое количество запущенных экземпляров), и задача контроллера состоит в том, чтобы согласовать и достичь желаемого целевого состояния. С этой точки зрения `Kubernetes` представляется в роли диспетчера распределенного состояния. Вы сообщаете ему параметры желаемого состояния экземпляра компонента, а он пытается достичь этого состояния и поддерживать его.

Можно ли как-то внедриться в этот процесс согласования без изменения кода `Kubernetes` и создать контроллер для наших конкретных потребностей?

Решение

В состав `Kubernetes` входит целая коллекция встроенных контроллеров, управляющих стандартными ресурсами, такими как `ReplicaSet`, `DaemonSet`, `StatefulSet`, `Deployment` или `Service`. Эти контроллеры работают под управлением диспетчера контроллеров, который развертывается (как отдельный

процесс или под) на главном узле. Контроллеры не знают о существовании друг друга. Они выполняют бесконечный цикл согласования, постоянно проверяя фактическое и желаемое состояние своих ресурсов и предпринимая соответствующие действия, чтобы приблизить фактическое состояние к желаемому.

Однако архитектура Kubernetes, управляемая событиями, позволяет подключать другие, нестандартные контроллеры. Эти контроллеры могут добавлять новые возможности и обрабатывать события подобно внутренним контроллерам. Все контроллеры по своей природе являются реактивными, они реагируют на события в системе и выполняют свои конкретные действия. В общем и целом процесс согласования состояния состоит из следующих основных этапов:

Наблюдение

Определение фактического состояния путем наблюдения за событиями, которые распространяет Kubernetes при изменении контролируемого ресурса.

Анализ

Выявление отличий от желаемого состояния.

Действие

Выполнение операций, необходимых для приведения текущего состояния в желаемое.

Например, контроллер ReplicaSet наблюдает за изменениями в ресурсе ReplicaSet, определяет, сколько подов должно быть запущено, и выполняет действия, посылая

определения подов в API Server. После этого внутренние механизмы Kubernetes производят запуск указанного пода на узле.

На рис. 22.1 показано, как контроллер регистрирует себя, подписываясь на события с целью обнаружения изменения состояния контролируемых ресурсов, наблюдает за их текущим состоянием и обращается к API Server (если это необходимо), чтобы поддержать фактическое состояние как можно ближе к желаемому.

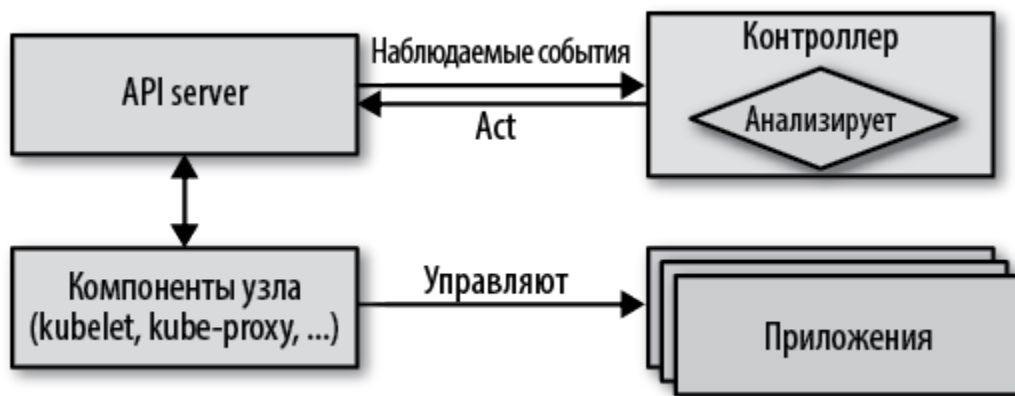


Рис. 22.1. Цикл наблюдение-анализ-действие

Контроллеры являются частью уровня управления в Kubernetes, и с самого начала было ясно, что они могут служить инструментом расширения платформы новыми аспектами поведения. Более того, они превратились в стандартный механизм расширения платформы и управления сложным жизненным циклом приложений. В результате родилось новое поколение более совершенных контроллеров под названием *Операторы*. С точки зрения эволюционного развития и сложности активные компоненты согласования состояния можно разделить на две группы:

Контроллеры

Реализуют простой процесс согласования, контролируя и воздействуя на стандартные ресурсы Kubernetes. Чаще всего эти контроллеры создаются с целью совершенствования поведения платформы и добавления новых возможностей.

Операторы

Реализуют сложный процесс согласования, который является основой паттерна *Operator* (Оператор), обслуживают определение нестандартных ресурсов *CustomResourceDefinition* (CRD). Как правило, операторы инкапсулируют сложную предметную логику и управляют полным жизненным циклом приложения. Мы подробно рассмотрим паттерн *Operator* (Оператор) в главе 23.

Как отмечалось выше, такое деление помогает постепенно внедрять новые идеи. В этой главе мы сосредоточимся на более простых контроллерах, а в следующей познакомимся с CRD и рассмотрим паттерн *Operator* (Оператор).

Чтобы исключить возможность одновременной обработки одних и тех же ресурсов, контроллеры реализуются с использованием паттерна *Singleton Service* (Служба-одиночка), описанного в главе 10. Большинство контроллеров развертываются с использованием ресурса *Deployment*, но с единственной репликой, потому что Kubernetes использует оптимистическую блокировку на уровне ресурсов для предотвращения проблем, связанных с параллельной обработкой при изменении объектов ресурсов. В конце концов, контроллер — это не что иное, как приложение, которое постоянно выполняется в фоновом режиме.

Поскольку сам фреймворк Kubernetes и клиентская библиотека для доступа к Kubernetes написаны на языке Go,

многие контроллеры тоже написаны на Go. Однако при желании контроллеры, посылающие запросы в Kubernetes API Server, можно писать на любом языке. Далее, в листинге 22.1, вы увидите контроллер, написанный на языке сценариев командной оболочки.

Наиболее просто реализуются контроллеры, расширяющие возможности Kubernetes управления ресурсами. Они оперируют теми же стандартными ресурсами и выполняют те же действия, что и внутренние контроллеры Kubernetes, но невидимые пользователям кластера. Контроллеры интерпретируют определения ресурсов и выполняют некоторые действия в соответствии со сложившимися условиями. Хотя они могут отслеживать и воздействовать на любой параметр в определении ресурса, лучше всего для этой цели подходят метаданные и карты конфигураций ConfigMap. Ниже перечислены некоторые соображения, которые следует учитывать при выборе места хранения данных контроллера:

Метки

Метки, как часть метаданных ресурса, доступны для анализа любым контроллерам. Они индексируются во внутренней базе данных, благодаря чему запросы с метками выполняются очень эффективно. Метки следует использовать всегда, когда требуется реализовать функциональность селектора (например, для выявления соответствующих подов в определениях Service или Deployment). Недостаток меток в том, что в метках можно использовать только буквенно-цифровые имена и значения с некоторыми ограничениями. Описание синтаксиса меток и набор допустимых символов можно найти в документации Kubernetes.

Аннотации

Аннотации являются отличной альтернативой меткам. Их следует использовать вместо меток, если значения не вписываются в синтаксические ограничения меток. Аннотации не индексируются, поэтому обычно они используются для представления информации, которая не используется в качестве ключей в запросах контроллеров. Еще одно преимущество аннотаций перед метками, кроме возможности представления произвольных метаданных, — они не оказывают отрицательного влияния на внутреннюю производительность Kubernetes.

Карты конфигураций ConfigMap

Иногда контроллерам нужна дополнительная информация, которую нельзя передать через метки и аннотации. В таких случаях для хранения определения целевого состояния можно использовать карты конфигураций ConfigMap, легко доступные контроллерам. Однако определения нестандартных ресурсов (CRD) намного лучше подходят для представления нестандартного описания целевого состояния и потому предпочтительнее, чем простые ConfigMap. Однако для регистрации CRD требуются повышенные привилегии на уровне кластера. Если у вас их нет, карты конфигураций ConfigMap останутся лучшей альтернативой CRD. Подробнее о CRD мы поговорим в главе 23 «Оператор».

Вот несколько примеров простых контроллеров, которые можно использовать для изучения способов реализации этого паттерна:

jenkins-x/exposecontroller

Этот контроллер (<http://bit.ly/2Ushlpy>) просматривает определения Service и, обнаружив в метаданных аннотацию с именем expose, автоматически создает объект Ingress для доступа к службе Service извне. Он также удаляет объект Ingress после удаления Service.

fabric8/configmapcontroller

Этот контроллер (<http://bit.ly/2uJ2FnI>) следит за изменениями в объектах ConfigMap и обновляет связанные с ними развертывания Deployment. Его можно использовать с приложениями, которые не способны наблюдать за ConfigMap и динамически обновляться при изменении конфигурации. Это особенно верно, когда ConfigMap отображается в переменные окружения или когда приложение не может быстро и надежно обновить свою конфигурацию на лету, без перезапуска. Реализация такого контроллера в виде сценария командной оболочки будет показана в листинге 22.2.

Оператор обновления контейнера Linux

Этот контроллер (<http://bit.ly/2uFcNgX>) перезагружает узел Kubernetes, обнаружив определенную аннотацию в узле.

Теперь рассмотрим конкретный пример: контроллер, состоящий из единственного сценария командной оболочки и наблюдающий за изменениями в ресурсах ConfigMap через Kubernetes API. Если снабдить ConfigMap аннотацией `k8spatterns.io/podDeleteSelector`, все поды, соответствующие этому значению аннотации, будут остановлены при изменении ConfigMap. Если эти поды

управляются посредством высокоуровневых ресурсов, таких как Deployment или ReplicaSet, они будут перезапущены с измененной конфигурацией.

Например, наш контроллер мог бы следить за изменением следующей карты конфигурации ConfigMap и перезапускать все поды с меткой app и значением webapp. Конфигурация из ConfigMap в листинге 22.1 используется нашим веб-приложением для определения приветственного сообщения.

Листинг 22.1. Ресурс ConfigMap для веб-приложения

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: webapp-config
  annotations:
    k8spatterns.io/podDeleteSelector:
"app=webapp" ❶
data:
  message: "Welcome to Kubernetes Patterns !"
```

❶ Аннотация, которая используется контроллером из листинга 22.2 как селектор для поиска подов, требующих перезапуска.

Этот ресурс ConfigMap будет проверяться нашим контроллером, реализованным в виде сценария командной оболочки. Полный исходный код вы найдете в репозитории Git. Контроллер запускает *зависающий* HTTP-запрос GET, чтобы открыть бесконечный поток HTTP-ответа, через который API Server передает события жизненного цикла.

События имеют форму простых объектов JSON. Контроллер анализирует события и определяет, содержит ли изменившийся ConfigMap нашу аннотацию. Получив событие, требующее реакции, контроллер останавливает все поды,

которые соответствуют селектору в значении аннотации. Давайте поближе рассмотрим работу контроллера.

Основу контроллера составляет цикл согласования, который принимает и обрабатывает события жизненного цикла ConfigMap, как показано в листинге 22.2.

Листинг 22.2. Сценарий контроллера

```
namespace=${WATCH_NAMESPACE:-default} ❶

base=http://localhost:8001 ❷
ns=namespaces/$namespace

curl -N -s $base/api/v1/${ns}/configmaps?
watch=true | \
while read -r event ❸
do
    # ...
done
```

❶ Пространство имен для наблюдения (или *default*, если не задано).

❷ Доступ к Kubernetes API осуществляется через прокси, реализованный с использованием паттерна *Ambassador* (Посредник) и действующий в том же поде.

❸ Цикл обработки событий изменения состояния ConfigMap.



Обратите внимание на параметр запроса `watch = true` в листинге 22.2. Этот параметр сообщает, что API Server должен оставить HTTP-соединение открытым и пересылать

через него события по мере их появления (этот прием также иногда называют *зависающий GET* или *Comet*). Цикл читает и обрабатывает поступающие события по отдельности.

Переменная окружения `WATCH_NAMESPACE` определяет пространство имен, в котором контроллер должен следить за обновлениями `ConfigMap`. Эту переменную можно инициализировать в описании развертывания `Deployment` самого контроллера. В данном примере, чтобы извлечь переменную окружения `WATCH_NAMESPACE` из пространства имен, в котором развернут контроллер, используется `Downward API`, описанный в главе 13 «Самоанализ», как показано в листинге 22.3.

Листинг 22.3. `WATCH_NAMESPACE` извлекается из текущего пространства имен

```
env:
```

```
- name: WATCH_NAMESPACE
```

```
  valueFrom:
```

```
    fieldRef:
```

```
      fieldPath: metadata.namespace
```

На основе пространства имен сценарий контроллера конструирует URL конечной точки `Kubernetes API` для наблюдения за `ConfigMap`.

Как видите, наш контроллер связывается с `Kubernetes API Server` через локальное соединение. Этот сценарий необязательно должен развертываться непосредственно на главном узле `Kubernetes API`, но тогда как он сможет работать, используя локальное соединение? Как вы, наверное, догадались, здесь на сцену выходит другой паттерн. Этот сценарий разворачивается в поде вместе с контейнером-посредником, который открывает порт 8001 на локальном хосте и связывает его с настоящей службой `Service` в `Kubernetes`. Более подробно о паттерне *Ambassador* (Посредник)

рассказывается в главе 17. Фактическое определение пода с этим посредником будет показано далее в этой главе.

Конечно, простое наблюдение за событиями — не самое надежное решение. Соединение может быть разорвано в любой момент, поэтому нужно предусмотреть возможность перезапустить цикл. Кроме того, события могут теряться, поэтому, действуя в промышленном окружении, контроллеры должны не только следить за событиями, но и время от времени запрашивать у API Server все текущее состояние и использовать его как новую основу. Но для демонстрации паттерна такого решения вполне достаточно.

В листинге 22.4 показана логика, выполняющаяся в цикле.

Листинг 22.4. Цикл согласования в контроллере

```
curl -N -s $base/api/v1/${ns}/configmaps?
watch=true | \
while read -r event
do
    type=$(echo "$event" | jq -r
'.type') ❶
    config_map=$(echo "$event" | jq -r
'.object.metadata.name')
    annotations=$(echo "$event" | jq -r
'.object.metadata.annotations')

    if [ "$annotations" != "null" ]; then
        selector=$(echo $annotations |
\
jq -r "\
to_entries
|\
.

```

```

        | \
                                select( .key      ==
\"k8spatterns.io/podDeleteSelector\") | \
        .value
            | \
            @uri
            \
        ")
    fi

    if [ $type = "MODIFIED" ] && [ -n "$selector"
]; then
        pods=$(curl -s $base/api/v1/${ns}/pods?
labelSelector=$selector | \
            jq -r .items[].metadata.name)

        for pod in $pods;
do
            curl -s -X DELETE
$base/api/v1/${ns}/pods/$pod
        done
    fi
done

```

❶ Извлечь из события тип и имя ConfigMap.

❷ Извлечь из ConfigMap все аннотации с ключом *k8spatterns.io/podDeleteSelector*. См. подробное описание этого выражения во врезке «Некоторые особенности jq» ниже.

❸ Если событие уведомляет об изменении ConfigMap и в нем имеется наша аннотация, найти все поды с метками, соответствующими этому селектору.

❹ Остановить все поды, соответствующие селектору.

Сначала сценарий извлекает тип события, определяющий произошедшее с ConfigMap. Далее извлекается ресурс ConfigMap и из него, с помощью jq, извлекаются аннотации. jq (<https://stedolan.github.io/jq/>) — отличный инструмент для анализа документов в формате JSON из командной строки, и данный сценарий предполагает его доступность в контейнере, где выполняется.

Если в ConfigMap есть аннотации, с помощью более сложного jq-запроса проверяется наличие среди них аннотации `k8spatterns.io/podDeleteSelector`. Цель этого запроса — преобразовать значение аннотации в селектор подов, который можно использовать в запросе к API на следующем шаге: аннотация `k8spatterns.io/podDeleteSelector: "app = webapp"` преобразуется в `app%3Dwebapp` — селектор подов. Это преобразование также выполняется с помощью jq, как описывается во врезке «Некоторые особенности jq» ниже.

Если сценарию удалось извлечь селектор, его можно сразу же и использовать, чтобы выбрать поды для остановки. Сначала выбираются все поды, которые соответствуют селектору, а затем останавливаются друг за другом прямыми вызовами API.

Этот контроллер в виде сценария командной оболочки, конечно, нельзя использовать в производстве (потому что, к примеру, цикл обработки событий может остановиться в любой момент), но он хорошо раскрывает основные идеи на небольшом объеме стандартного кода.

Остальная работа связана с созданием объектов ресурсов и образов контейнеров. Сам сценарий контроллера хранится в ConfigMap `config-watcher-controller`, и его легко можно поправить позже, если потребуется.

Некоторые особенности jq

Извлечение значения аннотации `k8spatterns.io/podDeleteSelector` и его преобразование в селектор подов выполняется с помощью `jq`. Это отличный инструмент командной строки для работы с документами JSON, но некоторые его идеи могут показаться немного необычными. Давайте рассмотрим, как работают выражения:

```
selector=$(echo $annotations | \
jq -r "\
  to_entries
    |\
[]
  |\
      select(.key ==
\"k8spatterns.io/podDeleteSelector\") |\
  .value
    |\
  @uri
  \
")
```

- Переменная `$annotations` содержит все аннотации в форме объекта JSON, в котором имена аннотаций играют роль свойств.

- Команда `to_entries` преобразует объект JSON вида `{ "a": "b" }` в массив с элементами `{ "key": "a", "value": "b" }`.

Подробности ищите в документации для `jq`.

- `.[]` выбирает элементы массива по одному.
- Из этих элементов выбираются только элементы с соответствующим ключом. После применения этого фильтра может остаться ноль или один элемент.
- Наконец, извлекается значение (`.value`), которое преобразуется командой `@uri` так, чтобы его можно было использовать как часть URI.

Это выражение преобразует структуру JSON, такую как

```
{
  "k8spatterns.io/pattern": "Controller",
  "k8spatterns.io/podDeleteSelector":
  "app=webapp"
}
```

в селектор `app%3Dwebapp`.

Развертывание Deployment для контроллера создает под с двумя контейнерами:

- Контейнер-посредник связывает порт 8001 на локальном хосте с Kubernetes API. Образ `k8spatterns/kubeapi-proxu` — это Alpine Linux с установленным локальным `kubectl` и запускается командой `kubectl proxu` с соответствующим сертификатом и токеном. Оригинальная версия `kubectl-proxu` была написана Марко Лукшей (Marko Lukša) и представлена в его книге «Kubernetes in Action»[18](#).
- Основной контейнер, в котором выполняется сценарий, содержащийся в только что созданном ConfigMap. Для его

создания используется базовый образ Alpine с установленными curl и jq.

Файлы *Dockerfile* с определениями образов k8spatterns/kubeapi-proxy и k8spatterns/curl-jq вы найдете в репозитории Git с примерами к книге.

Теперь, когда у нас есть образы для создания пода, осталось лишь развернуть контроллер с использованием Deployment. Основные разделы развертывания Deployment показаны в листинге 22.5 (полную версию ищите в репозитории с примерами).

Листинг 22.5. Описание развертывания контроллера

```
apiVersion: apps/v1
kind: Deployment
# ....
spec:
  template:
    # ...
    spec:
      serviceAccountName: config-watcher-
controller ❶
      containers:
        - name: kubeapi-
proxy ❷
          image: k8spatterns/kubeapi-proxy
        - name: config-
watcher ❸
          image: k8spatterns/curl-jq
          # ...
          command:
❹
```

```

    - "sh"
    - "/watcher/config-watcher-
controller.sh"
    volumeMounts:
    ⑤
    - mountPath: "/watcher"
      name: config-watcher-controller
    volumes:
    - name: config-watcher-
controller
    ⑥
    configMap:
      name: config-watcher-controller

```

① Учетная запись службы ServiceAccount с привилегиями, необходимыми для наблюдения за событиями и перезапуска подов.

② Контейнер-посредник для доступа к Kubeserver API через локальное соединение.

③ Основной контейнер со всеми инструментами и смонтированным сценарием контроллера.

④ Команда, запускающая сценарий контроллера.

⑤ Том, отображающийся в ConfigMap со сценарием.

⑥ Монтирование тома на основе ConfigMap в главный под.

Как видите, мы монтируем config-watcher-controller-script из ConfigMap, созданный ранее, и напрямую используем его в роли команды запуска в основном контейнере. Исключительно ради простоты мы опустили все проверки работоспособности и готовности, а также объявления с лимитами ресурсов. Также нам понадобилась учетная запись ServiceAccount config-watcher-controller с привилегиями, позволяющими следить за состоянием карт

конфигураций ConfigMap. Полные настройки безопасности вы найдете в репозитории с примерами.

Теперь посмотрим на этот контроллер в действии. Для этого возьмем простой веб-сервер, использующий значение переменной окружения в роли контента. Для этой цели базовый образ использует обычный nc (netcat). Файл *Dockerfile* с определением этого образа можно найти в нашем репозитории.

Развертывание HTTP-сервера реализовано с помощью ресурсов ConfigMap и Deployment, представленных в листинге 22.6.

Листинг 22.6. Deployment и ConfigMap с тестовым веб-приложением

```
apiVersion: v1
kind:
ConfigMap
❶
metadata:
  name: webapp-config
  annotations:
    k8spatterns.io/podDeleteSelector:
"app=webapp"   ❷
data:
  message: "Welcome to Kubernetes Patterns
!"           ❸
---
apiVersion: apps/v1
kind:
Deployment
❹
# ...
spec:
  # ...
```

```

template:
  spec:
    containers:
      - name: app
        image: k8spatterns/mini-http-
server      ⑤
        ports:
          - containerPort: 8080
        env:
          - name:
MESSAGE      ⑥
            valueFrom:
              configMapKeyRef:
                name: webapp-config
                key: message

```

- ① ConfigMap с данными для обслуживания.
- ② Аннотация, вызывающая перезапуск пода веб-приложения.
- ③ Сообщение, которое возвращает веб-приложение в HTTP-ответах.
- ④ Описание развертывания Deployment веб-приложения.
- ⑤ Простой образ с веб-сервером netcat.
- ⑥ Переменная окружения, содержимое которой используется как тело HTTP-ответа и извлекается из ресурса ConfigMap, находящегося под наблюдением контроллера.

На этом мы завершаем пример контроллера, наблюдающего за изменениями в ConfigMap и реализованного в виде сценария командной оболочки. Это, пожалуй, самый сложный пример в книге, но он достаточно ясно показывает,

что для создания простого контроллера не требуется писать много кода.

Очевидно, что для промышленного использования контроллеры должны программироваться на более мощных языках, обладающих лучшими средствами обработки ошибок и богатыми дополнительными возможностями.

Пояснение

Подводя итоги, можно сказать, что контроллер — это активный процесс согласования, наблюдающий за объектами, которые представляют желаемое состояние окружения. Он анализирует фактическое и желаемое состояния и посылает инструкции, пытаясь привести текущее состояние окружения к желаемому. Kubernetes реализует этот механизм во множестве своих внутренних контроллеров, и вы тоже можете использовать тот же механизм в своих собственных контроллерах. Мы увидели, что необходимо для того, чтобы создать свой контроллер, как он функционирует и расширяет возможности платформы Kubernetes.

Мы можем добавлять свои контроллеры благодаря модульной архитектуре Kubernetes, управляемой событиями. Такая архитектура естественным образом способствует созданию независимых и асинхронных контроллеров для ее расширения. Также существенным преимуществом является наличие четкой технической границы между самим фреймворком Kubernetes и любыми расширениями. Однако с асинхронной природой контроллеров связана одна проблема — их часто трудно отлаживать, потому что поток событий не всегда имеет простую структуру. Как следствие, нет простой возможности установить контрольные точки в контроллере, чтобы приостановить его и исследовать конкретную ситуацию.

В главе 23 вы познакомитесь с родственным паттерном *Operator* (Оператор), который основан на паттерне *Controller* (Контроллер) и предлагает еще более гибкий способ настройки операций.

Дополнительная информация

- Пример паттерна Контроллер (<http://bit.ly/2TWw6AW>).
- Создание контроллеров (<http://bit.ly/2HKlIWc>).
- Создание своего контроллера на Python (<https://red.ht/2HxC85a>).
- Подробное обсуждение контроллеров Kubernetes (<http://bit.ly/2ULdC3t>).
- Контроллер, открывающий доступ к службам (<https://github.com/jenkins-x/exposecontroller>).
- Контроллер для наблюдения за ConfigMap (<https://github.com/fabric8io/configmapcontroller>).
- Создание собственных контроллеров (<http://bit.ly/2TYgo9b>).
- Создание собственных контроллеров для Kubernetes (<http://bit.ly/2Cs1rS4>).
- Контроллер Contour Ingress (<https://github.com/heptio/contour>).
- Контроллер AppController (<https://github.com/Mirantis/k8s-AppController>).
- Набор символов, допустимых в метках (<http://bit.ly/2Q0td0M>).

- Kubectl-Proxy (<http://bit.ly/2FgearB>).

[18](#) Лукша Марко. Kubernetes в действии. М.: ДМК-Пресс. — *Примеч. пер.*

Глава 23. Оператор

Паттерн *Operator* (Оператор) — это тот же паттерн *Controller* (Контроллер), но использующий определения нестандартных ресурсов (*CustomResourceDefinition*, CRD) для представления практических знаний в конкретной предметной области в алгоритмической и автоматизированной форме. Паттерн *Operator* (Оператор) позволяет расширить паттерн *Controller* (Контроллер), описанный в предыдущей главе, и обеспечивает большую гибкость и выразительность.

Задача

В главе 22 «Контроллер» вы узнали, как расширить возможности платформы Kubernetes простым способом без внедрения в ее реализацию. Однако в некоторых случаях обычных нестандартных контроллеров недостаточно, потому что они могут контролировать только внутренние ресурсы Kubernetes. Иногда бывает желательно добавить новые понятия в платформу Kubernetes, требующие введения дополнительных предметных объектов. Допустим, в качестве решения для мониторинга мы выбрали Prometheus и хотим добавить его в Kubernetes четко определенным образом. Было бы просто замечательно, если бы у нас имелся ресурс Prometheus, описывающий конфигурацию и все детали развертывания средств мониторинга, подобный другим ресурсам Kubernetes. Также было бы хорошо иметь ресурсы, описывающие, какие службы следует охватить мониторингом (например, с помощью селектора меток).

В таких ситуациях используются определения нестандартных ресурсов (*CustomResourceDefinition*, CRD). Они

позволяют расширять Kubernetes API, добавляя нестандартные ресурсы в кластер Kubernetes и используя их, как если бы они были встроенными ресурсами. Нестандартные ресурсы вместе с паттерном *Controller* (Контроллером) для управления ими образуют паттерн *Operator* (Оператор).

Следующая цитата (<http://bit.ly/2Fjlx1h>) Джимми Зелински (Jimmy Zelinskie) как нельзя лучше описывает особенности паттерна *Operator* (Оператор):

Оператор — это контроллер Kubernetes, знающий две предметные области: Kubernetes и какую-то еще. Объединяя знания из обеих областей, он может автоматизировать задачи, обычно требующие участия оператора-человека, который понимает обе области.

Решение

В главе 22 «Контроллер» мы узнали о возможности эффективно реагировать на изменение состояния ресурсов, встроенных в Kubernetes. Теперь, понимая как работает первая половина паттерна *Operator* (Оператор), рассмотрим другую половину — представление нестандартных ресурсов в Kubernetes с использованием CRD.

Определения нестандартных ресурсов

Определения нестандартных ресурсов (CustomResourceDefinition, CRD) позволяют добавить в Kubernetes возможность управления понятиями из нашей предметной области. Нестандартные ресурсы управляются так же, как любые другие ресурсы, через Kubernetes API, и хранятся во внутреннем хранилище Etcd. Исторически

предшественниками CRD были сторонние ресурсы ThirdPartyResource.

Фактически упомянутый выше сценарий реализован с помощью новых нестандартных ресурсов в виде оператора CoreOS Prometheus, который обеспечивает бесшовную интеграцию Prometheus с Kubernetes. В листинге 23.1 показано определение CRD Prometheus, где также можно увидеть большинство параметров CRD.

Листинг 23.1. CustomResourceDefinition

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: prometheuses.monitoring.coreos.com ❶
spec:
  group: monitoring.coreos.com ❷
  names:
    kind: Prometheus ❸
    plural: prometheuses ❹
  scope: Namespaced ❺
  version: v1 ❻
  validation:
    openAPIV3Schema: ..... ❼
```

- ❶ Имя.
- ❷ Группа API, к которой принадлежит ресурс.
- ❸ Вид, используемый для идентификации экземпляров этого ресурса.
- ❹ Правило именования для создания формы множественного числа, используется для определения списка объектов.

⑤ Область видимости — может ли ресурс создаваться на уровне кластера или только в некотором пространстве имен.

⑥ Версия CRD.

⑦ Схема OpenAPI V3 для проверки (здесь не показана).

Также можно указать схему OpenAPI V3, чтобы Kubernetes мог проверить правильность определения нестандартного ресурса. В простых случаях схему можно опустить, но в промышленном окружении обязательно следует определить схему, чтобы ошибки конфигурации можно было обнаружить на самой ранней стадии.

Также Kubernetes позволяет указать в каждом CRD два возможных вложенных ресурса в поле `subresources`:

scale

С помощью этого свойства можно указать, как определяется количество реплик ресурса. В этом параметре можно указать путь в формате JSON для поиска желаемого количества реплик: путь к свойству, в котором хранится фактическое количество запущенных реплик, и необязательный путь к селектору меток, который можно использовать для поиска копий экземпляров ресурса. Обычно этот селектор меток можно не указывать, но он обязательно должен быть определен, если вы хотите использовать свой нестандартный ресурс с механизмом горизонтального масштабирования подов `HorizontalPodAutoscaler`, описанным в главе 24 «Эластичное масштабирование».

status

После установки этого свойства становится доступен новый вызов API, позволяющий изменять только статус. Этот

вызов API можно защитить отдельно и позволить с его помощью обновлять статус из-за пределов контроллера. С другой стороны, при обновлении ресурса в целом раздел `status` игнорируется точно так же, как для стандартных ресурсов Kubernetes.

Листинг 23.2 показывает, что пути к вложенным ресурсам определяются точно так же, как в обычных подах.

Листинг 23.2. Определение вложенных ресурсов в CustomResourceDefinition

```
kind: CustomResourceDefinition
# ...
spec:
  subresources:
    status: {}
    scale:
      specReplicasPath: .spec.replicas ❶
      statusReplicasPath: .status.replicas ❷
      labelSelectorPath: .status.labelSelector
```

❸

❶ Путь JSON к желаемому числу реплик.

❷ Путь JSON к числу активных реплик.

❸ Путь JSON к селектору меток для запроса количества активных реплик.

Определив CRD, легко можно создать такой ресурс, как в листинге 23.3.

Листинг 23.3. Нестандартный ресурс Prometheus

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus
spec:
  serviceMonitorSelector:
```

```
matchLabels:
  team: frontend
resources:
  requests:
    memory: 400Mi
```

Раздел `metadata`: имеет тот же формат и правила проверки, что и в любых других ресурсах Kubernetes. Раздел `spec`: содержит настройки, характерные для CRD, и Kubernetes проверяет его, используя правило из CRD.

Сами по себе нестандартные ресурсы не имеют большого практического значения без активных компонентов, работающих с ними. Чтобы сделать их полезными, необходим уже известный нам контроллер, наблюдающий за жизненным циклом этих ресурсов и действующий в соответствии с объявлениями в них.

Классификация контроллеров и операций

Прежде чем углубиться в реализацию паттерна *Operator* (Оператор), познакомимся поближе с классификацией контроллеров, операторов и особенно CRD. Основываясь на действиях, операторы можно разбить на следующие обширные группы:

CRD для установки

Предназначены для установки и запуска приложений на платформе Kubernetes. Типичным примером является CRD Prometheus, который можно использовать для установки и управления Prometheus.

Прикладные CRD

CRD этого типа используются для представления понятий предметной области и обеспечивают глубокую интеграцию приложений с Kubernetes, помогая объединить поведение Kubernetes с поведением конкретного приложения. Например, ServiceMonitor CRD используется оператором Prometheus для регистрации определенных служб Service в Kubernetes с целью наблюдения за сервером Prometheus. Оператор Prometheus заботится о соответствующей адаптации конфигурации сервера Prometheus.



Обратите внимание, что оператор может одновременно работать с разными типами CRD, как оператор Prometheus в данном случае. Граница между этими двумя категориями CRD весьма условна.

В нашей классификации паттернов *Controller* (Контроллер) и *Operator* (Оператор) оператор является¹⁹ контроллером, который использует определения нестандартных ресурсов CRD. Однако это отличие тоже несколько размыто из-за имеющихся вариаций.

Примером может служить контроллер, использующий ConfigMap как своеобразную замену CRD. Этот подход имеет смысл использовать в случаях, когда встроенных ресурсов Kubernetes недостаточно, а создание своего CRD невозможно. В таком случае ConfigMap может служить отличным промежуточным звеном, инкапсулирующим предметную логику. При использовании простого ConfigMap не нужно иметь права администратора кластера, необходимые для регистрации

CRD, и это большое преимущество. В некоторых кластерах просто нет возможности зарегистрировать такой CRD (например, в общедоступных кластерах, таких как OpenShift Online).

Заменив определение CRD картой конфигурации ConfigMap с предметными настройками, можно использовать подход *Наблюдение — Анализ — Действие*, но при этом вы лишитесь поддержки инструментов для CRD, таких как `kubectl get`, возможности проверки на уровне API Server и поддержки управления версиями API. Кроме того, вы не сможете оказать большого влияния на моделирование поля `status:` в ConfigMap, тогда как в CRD можно определить свою модель статуса, какую пожелаете.

Еще одно преимущество CRD — наличие подробной модели разрешений, основанной на типе CRD, которую можно настраивать индивидуально. Этот вид защиты RBAC невозможен, когда вся предметная конфигурация заключена в ConfigMap, потому что все ConfigMap в пространстве имен имеют одинаковую настройку разрешений.

С точки зрения реализации важно, что именно реализуется — контроллер, ограниченный использованием встроенными объектами Kubernetes, или у нас есть нестандартные ресурсы, которыми контроллер должен управлять. В первом случае у нас на выбор есть все типы, доступные в клиентской библиотеке Kubernetes. Во втором случае у нас нет готовой информации о типах и мы можем использовать подход к управлению ресурсами CRD без использования схемы или должны сами определить типы, возможно, на основе схемы OpenAPI, содержащейся в определении CRD. Поддержка типизированных CRD зависит от используемой клиентской библиотеки и инфраструктуры.

На рис. 23.1 показана классификация контроллеров и операторов, начиная с простых способов определения ресурсов и заканчивая сложными, при этом граница между контроллерами и операторами проводится по факту использования нестандартных ресурсов.

Для операторов в Kubernetes существуют точки расширения с еще более широкими возможностями. Когда определений CRD, управляемых Kubernetes, оказывается недостаточно для представления предметной области, можно расширить Kubernetes API собственным слоем агрегирования. Мы можем добавить ресурс APIService с собственной реализацией как новый URL-путь к Kubernetes API.

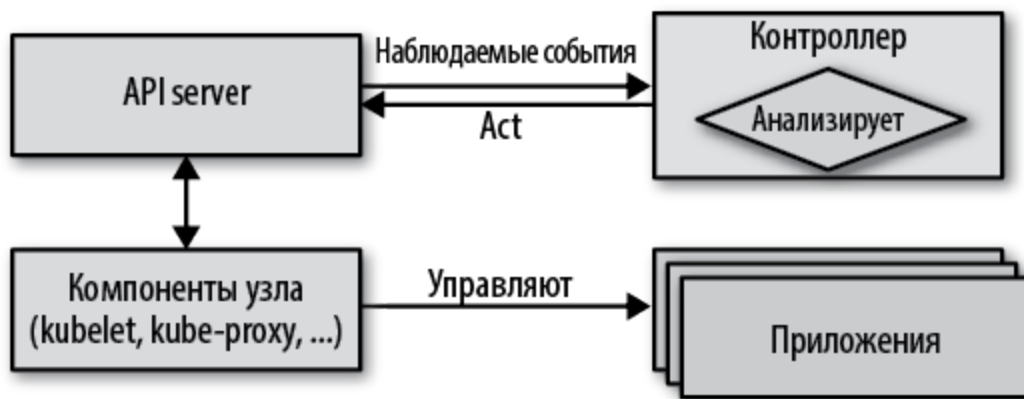


Рис. 23.1. Спектр контроллеров и операторов

Чтобы подключить данную службу Service с именем `custom-api-server` для поддержки пода с вашей службой, вы можете использовать ресурс, представленный в листинге 23.4.

Листинг 23.4. Слой агрегирования с нестандартным ресурсом APIService

```
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1alpha1.sample-api.k8spatterns.io
spec:
```

```
group: sample-api.k8spattterns.io
service:
  name: custom-api-server
version: v1alpha1
```

Кроме определений службы Service и пода, необходимы также некоторые дополнительные настройки безопасности для учетной записи ServiceAccount, под которой будет выполняться под.

После установки этой настройки все запросы к API Server (<https://<ip-адрес сервера API>/apis/sample-api.k8spatterns.io/v1alpha1/namespaces/<ns>/...>) будут направляться в реализацию нашей службы Service. Эта реализация должна обрабатывать все запросы, включая сохранение ресурсов, управляемых через этот API. Этот подход отличается от предыдущего случая использования CRD, где все управление нестандартными ресурсами осуществляется самой платформой Kubernetes.

В лице собственной реализации API Server вы получаете гораздо больше свободы, что позволяет не только наблюдать за событиями жизненного цикла ресурса. С другой стороны, этот подход требует реализовать гораздо больше логики, поэтому в типичных случаях использования обычно ограничиваются операторами, работающими с простыми CRD.

Подробное исследование возможностей API Server выходит за рамки этой главы. Более полную информацию вы найдете в официальной документации (<http://bit.ly/2uk7TWM>) и в законченном примере `sample-apiserver` (<http://bit.ly/2HJULSy>). Также можете использовать библиотеку `apiserver-builder` (<http://bit.ly/2JlhHEl>), которая поможет вам реализовать агрегирование с API Server.

А теперь посмотрим, как реализовать и развернуть оператор, управляющий нашими собственными определениями CRD.

Разработка и развертывание оператора

На момент написания этих строк (2019 г.) разработка операторов являлась активно развивающейся областью Kubernetes и имелось несколько инструментов и фреймворков для создания операторов. Вот три основных из них:

- CoreOS Operator Framework;
- Kubebuilder, разработанный непосредственно в SIG API Machinery Kubernetes;
- Metacontroller из Google Cloud Platform.

Мы кратко коснемся их далее, но помните, что все эти проекты довольно молоды и могут со временем измениться или даже объединиться.

Operator Framework

Operator Framework предлагает обширную поддержку разработки операторов на Golang и состоит из следующих компонентов:

- *Operator SDK* предлагает API высокого уровня для доступа к кластеру Kubernetes и упрощает создание проекта оператора.
- *Operator Lifecycle Manager* управляет выпуском и обновляет операторы и их определения CRD. Его можно рассматривать

как своеобразный «оператор операторов».

- *Operator Metering* упрощает добавление в операторы поддержки отчетов.

Мы не будем здесь вдаваться в детали Operator SDK, который продолжает активно развиваться, и остановимся на инструменте Operator Lifecycle Manager (OLM), предлагающем особенно ценную помощь при разработке операторов. Одна из проблем, связанных с ресурсами CRD, заключается в том, что их можно зарегистрировать только на уровне кластера, для чего необходимы привилегии администратора кластера²⁰. Обычные пользователи Kubernetes часто имеют возможность управлять всеми аспектами пространств имен, к которым им предоставлен доступ, но не могут просто использовать операторы без взаимодействия с администратором кластера.

OLM — это кластерная служба, действующая в фоновом режиме с привилегиями, разрешающими установку CRD. Вместе с OLM регистрируется выделенный ресурс CRD, называемый ClusterServiceVersion (CSV), позволяющий определить развертывание Deployment для оператора со ссылками на CRD, связанные с этим оператором. После создания такого ресурса CSV одна часть OLM ждет регистрации его и всех зависимых CRD. В этом случае OLM развертывает оператор, указанный в CSV. Другая часть OLM может использоваться для регистрации этих CRD от имени непривилегированного пользователя. Этот подход позволяет обычным пользователям кластера устанавливать свои операторы.

Kubebuilder

Kubebuilder — это проект группы SIG API Machinery с обширной документацией²¹. Подобно Operator SDK, он поддерживает создание проектов на Golang и управление несколькими CRD в одном проекте.

В отличие от Operator Framework, Kubebuilder работает напрямую с Kubernetes API, тогда как Operator SDK добавляет дополнительный слой абстракции поверх стандартного API, что упрощает его использование (но ценой утраты некоторых возможностей).

Поддержка установки и управления жизненным циклом оператора не так сложна, как в OLM из Operator Framework. Тем не менее оба проекта во многом совпадают и в конечном итоге могут прийти к общему знаменателю.

Metacontroller

Metacontroller сильно отличается от двух других инструментов разработки операторов. Он расширяет Kubernetes API своими функциями, реализующими общие возможности создания пользовательских контроллеров. Он действует подобно диспетчеру Kubernetes Controller Manager, запуская несколько контроллеров, которые определяются динамически с помощью особых ресурсов CRD, поддерживаемых Metacontroller. Иначе говоря, это контроллер, который вызывает службу с фактической логикой контроллера.

Описать Metacontroller можно также с точки зрения декларативного поведения. Ресурсы CRD дают возможность хранить новые типы в Kubernetes API, а Metacontroller позволяет легко определять поведение стандартных или пользовательских ресурсов декларативно.

Определяя контроллер с помощью Metacontroller, мы должны указать функцию, которая реализует бизнес-логику

нашего контроллера. *Metacontroller* берет на себя все хлопоты по взаимодействию с *Kubernetes API*, запускает цикл согласования от нашего имени и вызывает нашу функцию через заданную точку входа. Функции передается четко определенный набор данных, описывающий событие CRD. Так как функция должна возвращать значение, мы возвращаем определение ресурсов *Kubernetes*, которые требуется создать (или удалить) от имени нашей функции контроллера.

Такой способ, основанный на делегировании, позволяет писать функции на любом языке, способном обрабатывать HTTP и JSON, без всяких зависимостей от *Kubernetes API* или клиентских библиотек. Эти функции можно развертывать в *Kubernetes*, на стороне внешних поставщиков FaaS (*Functions-as-a-Service* — функция как услуга) или где-то еще.

Здесь у нас нет возможности подробно рассмотреть все варианты, но если вам требуется лишь дополнить *Kubernetes* простыми средствами автоматизации или управления и не нужны дополнительные функции, обратите внимание на *Metacontroller*, особенно если вы хотите реализовать свою бизнес-логику на языке, отличном от Go. В интернете можно найти примеры контроллеров, которые демонстрируют, как реализовать службу с состоянием, сине-зеленое развертывание, индексированное задание и службу-одиночку только с использованием *Metacontroller*.

Пример

Рассмотрим конкретный пример реализации паттерна *Operator* (Оператор). Здесь мы расширим пример из главы 22 «Контроллер» и определим CRD типа *ConfigWatcher*. Экземпляр этого CRD определяет ссылку на ресурс *ConfigMap* для наблюдения и то, какие поды должны перезапускаться при его изменении. Такой подход помогает устранить зависимость от

ConfigMap в подах, так как нам не нужно изменять сами ресурсы ConfigMap и добавлять в них аннотации, отвечающие за запуск логики. Кроме того, в упрощенном примере контроллера, основанного на аннотациях, мы можем подключить ConfigMap только к одному приложению. С использованием CRD возможны произвольные комбинации из ресурсов ConfigMap и подов.

В листинге 23.5 показано, как выглядит ресурс ConfigWatcher.

Листинг 23.5. Простой ресурс ConfigWatcher

```
kind: ConfigWatcher
apiVersion: k8spatterns.io/v1
metadata:
  name: webapp-config-watcher
spec:
  configMap: webapp-config ❶
  podSelector: ❷
    app: webapp
```

❶ Ссылка на ConfigMap для наблюдения.

❷ Селектор меток для выбора перезапускаемых подов.

В этом определении атрибут configMap ссылается на имя ConfigMap, за которым ведется наблюдение. Поле podSelector — это коллекция меток и их значений, которые идентифицируют перезапускаемые поды.

В листинге 23.6 показано определение типа этого ресурса в виде CRD.

Листинг 23.6. ConfigWatcher CRD

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
```

```

name: configwatchers.k8spatterns.io
spec:
  scope: Namespaced           ❶
  group: k8spatterns.io      ❷
  version: v1                 ❸
  names:
    kind: ConfigWatcher      ❹
    singular: configwatcher  ❺
    plural: configwatchers
  validation:
    openAPIV3Schema:         ❻
      properties:
        spec:
          properties:
            configMap:
              type: string
              description: "Name of the
ConfigMap"
            podSelector:
              type: object
              description: "Label selector for
Pods"
            additionalProperties:
              type: string

```

- ❶ Связано с пространством имен.
- ❷ Выделенная группа API.
- ❸ Начальная версия.
- ❹ Уникальный тип этого CRD.
- ❺ Метки ресурсов для использования с такими инструментами, как `kubectl`.

⑥ Схема OpenAPI V3 для этого CRD.

Чтобы наш оператор мог управлять ресурсами этого типа, необходимо привязать учетную запись ServiceAccount с соответствующими разрешениями для развертывания нашего оператора. С этой целью определим выделенную роль Role, как показано в листинге 23.7, которую позже используем в RoleBinding для ее привязки к ServiceAccount.

Листинг 23.7. Определение роли Role, открывающей доступ к ресурсу

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: config-watcher-crd
rules:
- apiGroups:
  - k8spatterns.io
  resources:
  - configwatchers
  - configwatchers/finalizers
  verbs: [ get, list, create, update, delete,
          deletecollection,
          watch ]
```

Объявив все эти CRD, можно определить ресурсы, как показано в листинге 23.5.

Чтобы придать этим ресурсам практическую ценность, нужно реализовать контроллер, который будет наблюдать за указанными в них ресурсами ConfigMap и перезапускать поды при их изменении.

Мы возьмем за основу сценарий контроллера из листинга 22.2 и изменим цикл обработки событий.

Наблюдая за изменениями в ConfigMap, вместо проверки конкретной аннотации мы будем запрашивать все ресурсы типа ConfigWatcher и проверять, указан ли изменившийся ConfigMap в атрибуте configMap:. В листинге 23.8 показан только цикл согласования. Полный пример вы найдете в репозитории Git, где также имеются подробные инструкции по установке этого оператора.

Листинг 23.8. Цикл согласования в контроллере WatchConfig

```
curl -Ns $base/api/v1/${ns}/configmaps?
watch=true | \ ❶
while read -r event
do
  type=$(echo "$event" | jq -r '.type')

  if [ $type = "MODIFIED" ];
then ❷

  watch_url="$base/apis/k8spatterns.io/v1/${n
s}/configwatchers"
  config_map=$(echo "$event" | jq -r
'.object.metadata.name')

  watcher_list=$(curl -s $watch_url | jq -r
'.items[]') ❸

  watchers=$(echo $watcher_list |
\ ❹
jq -r "select(.spec.configMap ==
\"$config_map\") |
.metadata.name")
```

```

        for watcher in watchers;
do
    label_selector=$(extract_label_selector
$watcher)
    delete_pods_with_selector
"$label_selector"
done
fi
done

```

❶ Запуск потока событий для наблюдения за изменениями в ConfigMap в заданном пространстве имен.

❷ Проверять только события MODIFIED.

❸ Получить список всех установленных ресурсов ConfigWatcher.

❹ Извлечь список всех элементов ConfigWatcher, ссылающихся на данный ConfigMap.

❺ Для каждого найденного ConfigWatcher остановить указанные поды через селектор. Логика для формирования селектора меток, а также остановки подов здесь опущена. Полную реализацию вы найдете в репозитории Git с примерами.

Этот контроллер можно протестировать с тем же примером веб-приложения в нашем репозитории Git, что и контроллер из главы 22 «Контроллер». Единственное его отличие в том, что здесь для настройки приложения используется неаннотированный ConfigMap.

Наш оператор на основе сценария командной оболочки получился достаточно функциональным, но также очевидно, что он по-прежнему довольно прост и не предусматривает обработку крайних случаев или ошибок. При желании вы

сможете найти еще много интересных примеров промышленного уровня.

В списке Awesome Operators (<http://bit.ly/2Ucjs0J>) вы найдете множество действующих операторов, которые основаны на идеях, описанных в этой главе. Мы уже видели, как оператор Prometheus может управлять экземплярами Prometheus. Другой пример оператора, написанного на Golang, называется Etcd Operator и предназначен для управления хранилищем пар ключ/значение и автоматизации таких задач, как резервное копирование и восстановление базы данных.

Если вы ищете оператор, написанный на Java, обратите внимание на *Strimzi Operator*, который послужит вам отличным примером оператора, управляющего сложной системой обмена сообщениями в Kubernetes, такой как Apache Kafka. Еще одним интересным инструментом для разработки операторов на Java является *JVM Operator Toolkit*, который обеспечивает основу для создания операторов на Java и других языках JVM, таких как Groovy и Kotlin, а также содержит набор примеров.

Пояснение

Мы увидели, как можно расширить возможности платформы Kubernetes, но имейте в виду, что операторы не являются универсальным решением на все случаи жизни. Прежде чем приступать к созданию своего оператора, внимательно изучите свой случай, чтобы определить, соответствует ли он парадигме Kubernetes.

Во многих случаях вполне достаточно обычного контроллера, работающего со стандартными ресурсами. Преимущество этого подхода заключается в том, что он не требует полномочий администратора кластера для регистрации

CRD, но он имеет также свои ограничения, когда речь заходит о безопасности или проверке.

Операторы хорошо подходят для моделирования предметной логики, сочетающейся с декларативным способом обработки ресурсов в Kubernetes и использованием реактивных контроллеров.

В частности, операторы и CRD хорошо подходят в любой из следующих ситуаций, когда:

- требуется тесная интеграция с существующими инструментами Kubernetes, такими как `kubectl`;
- проект находится на старте и вы можете спроектировать приложение с нуля;
- очевидна выгода от концепций Kubernetes, таких как пути ресурсов, группы API, версии API и особенно пространства имен;
- необходима хорошая клиентская поддержка для доступа к API с часами, аутентификацией, авторизацией на основе ролей и селекторами метаданных.

Если ваш пользовательский сценарий соответствует этим критериям, но нужна большая гибкость в том, как пользовательские ресурсы могут быть реализованы и сохранены, подумайте об использовании пользовательского сервера API. При этом не стоит рассматривать точки расширения Kubernetes в качестве панацеи.

Если ваш случай использования не является декларативным, если данные для управления не подходят для модели ресурсов Kubernetes или вам не нужна тесная интеграция в платформу, вероятно, лучше всего написать свой

автономный API и опубликовать его с помощью классического объекта Service или Ingress.

В документации Kubernetes также есть глава с предложениями по использованию контроллера, оператора, агрегации API или пользовательской реализации API.

Дополнительная информация

- Operator Example (<http://bit.ly/2HvflkV>).
- Фреймворк оператора (<http://bit.ly/2CKLYN1>).
- OpenAPI V3 (<http://bit.ly/2Tluk82>).
- Kubebuilder (<http://bit.ly/2l8w9mz>).
- Клиентские библиотеки Kubernetes (<http://bit.ly/2Sh1XYk>).
- Metacontroller (<https://metacontroller.app/>).
- JVM Operator Toolkit (<https://github.com/jvm-operators>).
- Расширение API Kubernetes с помощью CustomResourceDefinitions (<http://bit.ly/2uk6lq5>).
- Awesome Operators в действии (<http://bit.ly/2Ucjs0J>).
- Custom Resources и API Server Aggregations (<http://bit.ly/2FrR6l>).
- Сравнение Kubebuilder, Operator Framework и Metacontroller (<http://bit.ly/2Fp04Ug>).

- TPR мертв! Kubernetes 1.7 включает CRD (<http://bit.ly/2FQnCSA>).
- Генерация кода для пользовательских ресурсов (<https://red.ht/2HIS9Er>).
- Простой Operator в Go (<http://bit.ly/2UppsTN>).
- Prometheus (<http://bit.ly/2HICRjT>).
- Etcd (<http://bit.ly/2JTz8SK>).
- Memhog (<https://github.com/secat/memhog-operator>).

¹⁹ Слово «является» (<https://en.wikipedia.org/wiki/Is-a>) подчеркивает отношение наследования между паттернами *Operator* (Оператор) и *Controller* (Контроллер), где *Operator* (Оператор) обладает всеми особенностями *Controller* (Контроллер) и добавляет свои.

²⁰ Это ограничение может быть снято в будущем, так как уже существуют планы регистрации определений CRD в рамках пространств имен.

²¹ Специальные группы по интересам (Special Interest Groups, SIG) — это способ организации сообщества Kubernetes по функциональным областям. Список всех существующих групп SIG в проекте Kubernetes можно найти в репозитории GitHub (<https://github.com/kubernetes-sigs>).

Глава 24. Эластичное масштабирование

Паттерн *Elastic Scale* (Эластичное масштабирование) охватывает масштабирование приложений в нескольких измерениях: горизонтальное масштабирование путем коррекции количества реплик пода; вертикальное масштабирование путем коррекции требований к ресурсам для подов; и масштабирование самого кластера путем изменения количества узлов. Все эти действия можно выполнить вручную, но в этой главе мы посмотрим, как то же самое может делать Kubernetes автоматически, в зависимости от нагрузки.

Задача

Kubernetes автоматизирует управление распределенными приложениями, состоящими из большого количества неизменяемых контейнеров, поддерживая желаемое их состояние, выраженное декларативно. Однако, учитывая непостоянный характер многих рабочих нагрузок, которые часто меняются со временем, непросто понять, как должно выглядеть желаемое состояние. Точное определение, сколько ресурсов потребуется контейнеру и сколько реплик службы должно быть запущено в данный момент для выполнения соглашений об уровне обслуживания, требует времени и усилий. К счастью, Kubernetes позволяет легко изменять объем ресурсов контейнера, число реплик службы или узлов в кластере. Такие изменения могут происходить вручную или автоматически, в соответствии с определенными правилами.

Kubernetes может не только следовать фиксированным настройкам подов и кластера, но также следить за уровнем нагрузки и событиями, связанными с изменением объемов

ресурсов, анализировать текущее состояние и масштабироваться для достижения желаемой производительности. Эта способность наблюдать позволяет Kubernetes адаптироваться и обретать эластичность, опираясь на фактические показатели использования, а не на ожидаемые факторы. Давайте рассмотрим разные способы, которыми можно достичь такого поведения, и то, как объединить разные методы масштабирования.

Решение

Есть два основных подхода к масштабированию любых приложений: горизонтальное и вертикальное. Под *горизонтальным масштабированием* в мире Kubernetes подразумевается создание большего количества реплик пода. Под *вертикальным масштабированием* подразумевается увеличение объема ресурсов для контейнеров, управляемых подами. На бумаге все выглядит просто, но определить конфигурацию приложения для автоматического масштабирования на общей облачной платформе так, чтобы не повлиять на другие службы и на сам кластер, часто можно только методом проб и ошибок. Как всегда, Kubernetes предлагает множество средств и методов, помогающих найти лучшие настройки для приложений, и мы кратко рассмотрим их здесь.

Горизонтальное масштабирование вручную

Ручное масштабирование, как нетрудно догадаться, заключается в том, что оператор-человек посылает команды Kubernetes. Этот подход можно использовать в отсутствие автоматического масштабирования или для постепенного поиска оптимальной конфигурации приложения,

соответствующей медленно меняющейся нагрузке, в течение длительных периодов. Преимущество ручного подхода заключается в том, что он допускает упреждающие, а не только реактивные изменения: зная периодичность и ожидаемую нагрузку на приложение, его можно масштабировать заранее, а не реагируя, например, на уже возросшую нагрузку с использованием средств автоматического масштабирования. Ручное масштабирование может выполняться в двух стилях.

Императивное масштабирование

Контроллер, такой как ReplicaSet, отвечает за постоянное выполнение заданного количества экземпляров пода. Благодаря этому, чтобы масштабировать под, достаточно изменить количество желаемых реплик. Масштабировать наше развертывание Deployment с именем random-generator до четырех экземпляров можно одной командой, как показано в листинге 24.1.

Листинг 24.1. Изменение числа реплик развертывания Deployment из командной строки

```
kubectl scale random-generator --replicas=4
```

После такого изменения контроллер ReplicaSet может запустить дополнительные поды или, если число действующих подов больше желаемого, остановить избыточные.

Декларативное масштабирование

Масштабирование с использованием команды scale выполняется тривиально просто и удобно для быстрого реагирования в чрезвычайных ситуациях, но этот подход не сохраняет настройки вне кластера. Обычно все приложения для Kubernetes хранят определения своих ресурсов в системе управления версиями, включая число реплик. Воссоздание

ReplicaSet из исходного определения приведет к изменению числа реплик до прежнего уровня. Чтобы избежать такого отклонения конфигурации и обеспечить обратное распространение изменений, рекомендуется декларативно изменять желаемое количество реплик в ReplicaSet или некотором другом определении и применять изменения к Kubernetes, как показано в листинге 24.2.

Листинг 24.2. Использование развертывания Deployment для декларативной настройки числа реплик

```
kubectl apply -f random-generator-deployment.yaml
```

Мы можем масштабировать ресурсы, управляющие несколькими подами, такие как ReplicaSet, Deployment и StatefulSet. Обратите внимание на асимметричное поведение при масштабировании StatefulSet с постоянным хранилищем. Как рассказывалось в главе 11 «Служба с состоянием», если в StatefulSet имеется элемент `.spec.volumeClaimTemplates`, он будет создавать PVC при масштабировании вверх, но не будет удалять при масштабировании вниз, чтобы предотвратить удаление хранилища.

Еще одним ресурсом Kubernetes, доступным для масштабирования, но следующим другим соглашениям об именовании, является ресурс задания Job, который был описан в главе 7 «Пакетное задание». Масштабирование задания с целью параллельного выполнения нескольких экземпляров одного и того же пода осуществляется изменением поля `.spec.parallelism`, а не `.spec.replicas`. Однако это дает тот же семантический эффект: увеличение пропускной способности путем запуска дополнительных обрабатывающих подов, которые действуют как одна логическая единица.



Для описания полей в этой книге используется формат определения путей JSON. Например, имя `.spec.replicas` соответствует полю `replicas` в разделе `spec` объявления ресурса.

Оба стиля масштабирования вручную (императивный и декларативный) предполагают, что человек будет наблюдать или предвидеть изменение нагрузки на приложение, принимать решение о направлении и величине масштабирования и применять это решение к кластеру. Однако эти стили не подходят для приложений с динамической нагрузкой, которая часто меняется и требует постоянной адаптации. Давайте посмотрим далее, как можно автоматизировать принятие решений о масштабировании.

Автоматическое горизонтальное масштабирование

Многие рабочие нагрузки имеют динамическую природу, меняясь со временем и затрудняя определение фиксированных настроек масштабирования. Но облачные технологии, такие как Kubernetes, позволяют создавать приложения, легко адаптирующиеся к изменяющимся нагрузкам. Поддержка автоматического масштабирования в Kubernetes позволяет определять переменную емкость приложения вместо фиксированной, которая обеспечивает достаточную пропускную способность для обслуживания другой нагрузки. Проще всего такое поведение реализовать с использованием `HorizontalPodAutoscaler` (HPA) для горизонтального масштабирования числа подов.

Определение HPA для развертывания Deployment random-generator можно создать с помощью команды в листинге 24.3. Чтобы это определение HPA имело какой-либо эффект, важно, чтобы при развертывании было объявлено ограничение .spec.resources.requests для процессора, как описано в главе 2 «Предсказуемые требования». Также важно, чтобы в кластере был включен сервер метрик, который собирает данные об использовании ресурсов.

Листинг 24.3. Создание определения HPA из командной строки

```
kubectl autoscale deployment random-generator -  
-cpu-percent=50 --min=1 --max=5
```

Эта команда создаст определение HPA, показанное в листинге 24.4.

Листинг 24.4. Определение HPA

```
apiVersion: autoscaling/v2beta2  
kind: HorizontalPodAutoscaler  
metadata:  
  name: random-generator  
spec:  
  minReplicas: 1 ❶  
  maxReplicas: 5 ❷  
  scaleTargetRef: ❸  
    apiVersion: extensions/v1beta1  
    kind: Deployment  
    name: random-generator  
  metrics:  
  - resource:  
    name: cpu  
    target:  
      averageUtilization: 50 ❹
```

type: Utilization
type: Resource

- ❶ Минимальное число всегда выполняющихся подов.
- ❷ Максимальное число подов, которое может быть достигнуто при масштабировании вверх.
- ❸ Ссылка на объект, связанный с этим определением HPA.
- ❹ Уровень использования процессора в процентах от запрошенного в подах. Например, если в поде параметр `.spec.resources.requests.cpu` имеет значение `200m`, масштабирование вверх произойдет, когда средняя величина использования процессора превысит `100m (= 50%)`.



В листинге 24.4 для настройки HPA используется версия API ресурса `v2beta2`. Эта версия находится в активной разработке и является расширением версии `v1`. Версия `v2` предлагает намного больше критериев, чем загрузка процессора: например, потребление памяти или нестандартные метрики для конкретного приложения. Командой `kubectl get hpa.v2beta2.autoscaling -o yaml` легко можно преобразовать ресурс HPA `v1`, созданный командой `kubectl autoscale`, в ресурс `v2`.

Следуя этому определению, контроллер HPA будет стремиться сохранить среднее потребление процессора на уровне `50%` от значения, затребованного в `.spec.resources.requests`, изменяя количество запущенных экземпляров пода в диапазоне от одного до пяти. Такое определение HPA можно применить к любым ресурсам,

которые поддерживают подресурс `scale`, такие как `Deployment`, `ReplicaSet` и `StatefulSet`, но вы должны учитывать побочные эффекты. Развертывания `Deployment` создают новые наборы реплик во время обновлений, но не копируют никаких определений HPA. Если применить определение HPA к `ReplicaSet`, который управляется развертыванием `Deployment`, оно не будет скопировано в новые `ReplicaSet` и просто потеряется. Лучше всего применять HPA к абстракции развертывания `Deployment` более высокого уровня, которая сохраняет и применяет HPA к новым версиям `ReplicaSet`.

Теперь посмотрим, как определение HPA может заменить оператора-человека и обеспечить автоматическое масштабирование. В общем случае контроллер HPA непрерывно выполняет следующие действия:

1. Извлекает метрики подов, которые подлежат масштабированию согласно определению HPA. Метрики извлекаются не напрямую из подов, а из `Kubernetes Metrics API`, возвращающего агрегированные метрики (и даже пользовательские и внешние метрики, если выполнены соответствующие настройки). Метрики уровня подов извлекаются из `Metrics API`, а все остальные — из `Kubernetes Custom Metrics API`.
2. Вычисляет необходимое количество реплик на основе текущего и целевого значения метрики. Вот упрощенная версия формулы:

$$\text{требуемоеЧислоРеплик} = \left\lceil \text{текущееЧислоРеплик} \times \frac{\text{текущееЗначениеМетрики}}{\text{желаемоеЗначениеМетрики}} \right\rceil.$$

Например, если в настоящий момент выполняется единственный под с текущим значением метрики

использования процессора, равным 90% от запрошенного²², а желаемое значение составляет 50%, тогда количество реплик будет удвоено, так как $\left\lceil 1 \times \frac{90}{50} \right\rceil = 2$. В действительности используется более сложная реализация, потому что она должна учитывать несколько выполняющихся экземпляров пода, несколько типов метрик и множество крайних случаев и колеблющихся значений. Например, если задано несколько метрик, тогда контроллер НРА оценит каждую метрику отдельно и из всех результатов выберет наибольший. Конечный результат всех вычислений — это одно целое число, представляющее количество желаемых реплик, которое поможет удержать измеренное значение метрики ниже желаемого порогового значения.

В поле `replicas` в определении ресурса, подлежащего автоматическому масштабированию, будет записано это вычисленное число, после чего другие контроллеры выполнят свою работу по достижению и поддержанию нового требуемого состояния. На рис. 24.1 показано, как работает НРА: осуществляет мониторинг метрик и изменяет число реплик соответственно.

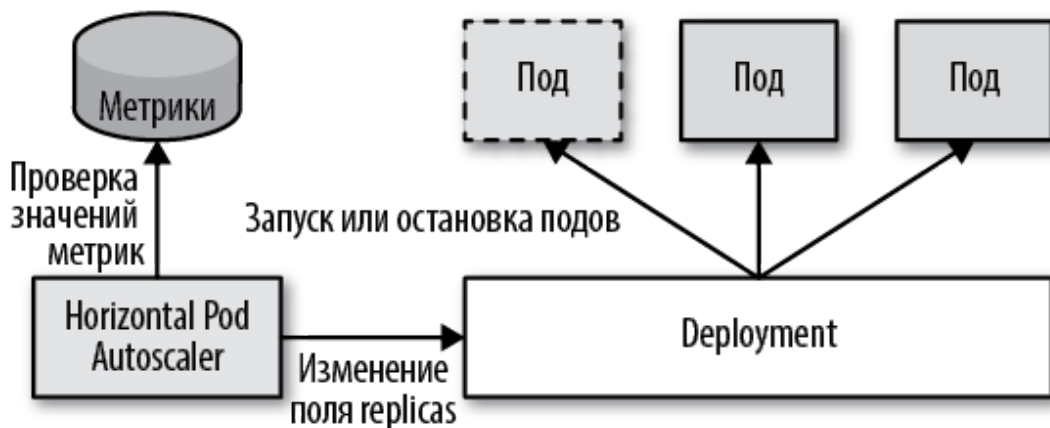


Рис. 24.1. Механизм автоматического горизонтального масштабирования подов

Автоматическое масштабирование — это область Kubernetes со множеством мелких деталей, которые продолжают быстро развиваться, и каждая из них может оказать существенное влияние на общее поведение механизма автоматического масштабирования. Поэтому здесь мы не будем подробно обсуждать все детали, но в разделе «Дополнительная информация» вы найдете ссылки на самую последнюю актуальную информацию по этому вопросу.

В целом поддерживаются следующие типы метрик:

Стандартные метрики

Эти метрики объявлены с параметром `.spec.metrics.resource[:].type`, содержащим значение `Resource`, и представляют метрики использования ресурсов, таких как процессор и память. Они доступны для любого контейнера в любом кластере под одними и теми же именами. Их значения можно определять в процентах, как это сделано в предыдущем примере, или в абсолютных значениях. В обоих случаях значения описывают гарантированный объем ресурса, то есть объем в поле `requests` контейнера, а не `limits`. Это самые простые в использовании типы метрик и обычно поддерживаются сервером метрик или компонентами Heapster, которые можно запускать как дополнения кластеров.

Нестандартные метрики

Эти метрики с параметром `.spec.metrics.resource[:].type`, содержащим значение `Object` или `Pod`, требуют расширенных настроек мониторинга кластера, которые могут отличаться для

разных кластеров. Нестандартная метрика с типом Pod, как можно догадаться, описывает метрику, характерную для пода, тогда как метрика с типом Object может описывать любой другой объект. Нестандартные метрики обслуживаются агрегирующим API Server с точкой входа `custom.metrics.k8s.io` и поддерживаются различными адаптерами метрик, такими как Prometheus, Datadog, Microsoft Azure и Google Stackdriver.

Внешние метрики

К этой категории относятся метрики, которые описывают ресурсы, не являющиеся частью кластера Kubernetes. Например, у вас может быть под, принимающий сообщения от облачной службы очередей. Очень часто в таком сценарии бывает желательно масштабировать количество подов-потребителей в зависимости от глубины очереди. Такие метрики поддерживаются внешними плагинами метрик и имеют много общего с нестандартными метриками.

Правильно настроить автоматическое масштабирование очень непросто. Для этого придется немного поэкспериментировать с настройками. Ниже перечислены основные аспекты, которые следует учитывать при настройке НРА:

Выбор метрик

Одним из самых, пожалуй, важных решений, касающихся автоматического масштабирования, является выбор метрик. Чтобы добиться максимальной отдачи от НРА, между значением метрики и количеством реплик пода

должна существовать прямая корреляция. Например, если выбранная метрика определяет количество запросов в секунду (например, HTTP-запросов в секунду), увеличение количества подов приведет к уменьшению среднего числа запросов, потому что запросы будут передаваться большему количеству подов. То же верно для метрики, определяющей использование процессора, потому что существует прямая корреляция между частотой запросов и использованием процессора (увеличение числа запросов ведет к увеличению использования процессора). Другие метрики, такие как потребление памяти, не имеют такой прямой корреляции. Проблема метрики потребления памяти заключается в том, что если служба потребляет определенный объем памяти, запуск большего числа экземпляров пода, скорее всего, не приведет к снижению потребления памяти, если приложение ничего не знает о других экземплярах и не имеет механизмов для перераспределения и освобождения своей памяти. Если память не освобождается и это отражается в метриках, НРА будет создавать все больше и больше подов, чтобы уменьшить объем потребляемой памяти, пока не достигнет верхнего порога на число реплик, что, вероятно, не является желаемым поведением. Поэтому выбирайте метрики, прямо (предпочтительно линейно) зависящие от количества подов.

Предотвращение частого изменения числа реплик

Механизм автоматического масштабирования НРА применяет разные методы, чтобы избежать быстрого выполнения противоречивых решений, которые могут приводить к изменению количества реплик при нестабильной нагрузке. Например, во время масштабирования вверх НРА игнорирует пиковые нагрузки

на процессор, обусловленные инициализацией подов, обеспечивая гладкую реакцию на увеличение нагрузки. Во время масштабирования вниз, чтобы избежать неоправданного уменьшения числа реплик в короткие периоды провалов, контроллер учитывает все рекомендации по масштабированию в течение настраиваемого временного окна и выбирает самую высокую рекомендацию, имевшую место в течение этого окна. Все это улучшает устойчивость НРА к случайным колебаниям метрик.

Отложенная реакция

Масштабирование, инициируемое изменением значения метрики, — это многостадийный процесс, в который вовлечено несколько компонентов Kubernetes. Первый — это агент cAdvisor, действующий в каждом контейнере, и регулярно собирающий метрики для Kubelet. Второй — сервер метрик, извлекающий метрики из Kubelet через регулярные интервалы. Наконец, периодически запускается контроллер НРА, который анализирует собранные метрики. Формула масштабирования НРА вводит некоторую задержку, чтобы предотвратить частое применение противоречивых решений (как описано в предыдущем пункте). Вся эта деятельность накапливается в задержке между причиной и реакцией масштабирования. Увеличение задержки настройкой этих параметров делает НРА менее отзывчивым, а уменьшение увеличивает нагрузку на платформу. Настройка Kubernetes для балансировки ресурсов и производительности — это постоянный процесс обучения.

Knative Serving

Проект Knative Serving (с которым мы познакомимся в разделе «Knative Build» главы 25) предлагает еще более продвинутые методы горизонтального масштабирования. К их числу относится поддержка «масштабирования до нуля», когда число подов, поддерживающих службу Service, может быть уменьшено до нуля и увеличивается, только когда происходит определенное событие, например входящий запрос. В Knative эта возможность реализована поверх «сетки служб» Istio, которая, в свою очередь, предлагает прозрачные службы внутренней маршрутизации для подов. Knative Serving образует основу для бессерверной архитектуры, обладающей еще более гибкими и быстрыми средствами горизонтального масштабирования, превосходящими стандартные механизмы Kubernetes.

Подробное обсуждение Knative Serving выходит за рамки этой книги, так как это еще очень молодой проект, заслуживающий отдельной книги. Тем не менее в разделе «Дополнительная информация» вы найдете дополнительные ссылки на ресурсы Knative.

Автоматическое вертикальное масштабирование

Горизонтальное масштабирование предпочтительнее вертикального, потому что менее разрушительно, особенно для служб без состояния. Это не относится к службам с состоянием, для которых вертикальное масштабирование может оказаться предпочтительнее. В числе других сценариев применения вертикального масштабирования можно назвать

настройку фактических потребностей службы в ресурсах на основе фактической нагрузки. Мы выяснили, почему определение правильного количества реплик пода может быть затруднено или даже невозможно, когда нагрузка меняется с течением времени. Вертикальное масштабирование тоже имеет подобные проблемы, связанные с определением правильных значений для параметров `requests` и `limits` контейнера. Механизм автоматического вертикального масштабирования подов в Kubernetes (Vertical Pod Autoscaler, VPA) решает эти проблемы за счет автоматизации процесса настройки и распределения ресурсов на основе информации о фактическом их потреблении.

Как мы видели в главе 2 «Предсказуемые требования», каждый контейнер в поде может определять свои параметры `requests` с требованиями к процессорному времени и памяти, что влияет на планирование подов. Параметры `requests` и `limits` в некотором смысле определяют контракт между подом и планировщиком, который обеспечивает гарантированный объем ресурсов или отказывается запланировать под. Слишком низкие требования к объему памяти могут привести к тому, что узлы будут плотно забиты подами, что, в свою очередь, может привести к ошибкам нехватки памяти или остановке действующих подов из-за нехватки памяти. Слишком низкие требования к процессорному времени могут привести к нехватке вычислительных ресурсов и низкой эффективности приложений. С другой стороны, если запросить избыточно большой объем ресурсов, это приводит к напрасному их расходованию. Важно определять параметры `requests` с требованиями к ресурсам как можно точнее, потому что это влияет на эффективность расходовании ресурсов кластера и

горизонтального масштабирования. Давайте посмотрим, как VPA помогает решить эту проблему.

Рассмотрим определение VPA из листинга 24.5, чтобы поближе познакомиться с автоматическим вертикальным масштабированием подов в кластере с поддержкой VPA и сервером метрик.

Листинг 24.5. VPA

```
apiVersion: pos.autoscaling.k8s.io/v1alpha1
kind: VerticalPodAutoscaler
metadata:
  name: random-generator-vpa
spec:
  selector:
    matchLabels:
      app: random-generator
  updatePolicy:
    updateMode: "Off"
```

❶ Селектор меток для идентификации подов.

❷ Политика обновления определяет, как механизм VPA будет применять изменения.

Определение VPA включает следующие важные элементы:

Селектор меток

Определяет поды, подлежащие масштабированию.

Политика обновления

Определяет, как механизм VPA будет применять изменения. Режим `Initial` позволяет выделять запрошенные ресурсы

только во время создания пода, но не позже. Режим по умолчанию Auto позволяет выделять запрошенные ресурсы во время создания пода, а также вытеснять и снова планировать поды при изменении требований. Значение Off отключает автоматическое применение изменившихся требований подов, но позволяет определять предлагаемые значения. Это своего рода пробный прогон для определения нужного размера контейнера, но без непосредственного его применения.

Определение VPA также может иметь политику выделения ресурсов, которая влияет на то, как VPA вычисляет рекомендуемый объем ресурсов (например, путем установки для каждого контейнера нижней и верхней границ).

В зависимости от значения параметра `.spec.updatePolicy.updateMode`, VPA вовлекает в работу разные системные компоненты. Все три компонента VPA — механизмы рекомендаций, согласования требований и обновления — действуют независимо и могут заменяться альтернативными реализациями. Интеллектуальный механизм рекомендаций создавался с учетом опыта разработки системы Google Borg. Текущая реализация анализирует фактическое использование ресурсов контейнером под нагрузкой в течение определенного периода (по умолчанию восемь дней), создает гистограмму и выбирает значение, соответствующее наибольшему процентилю за этот период. Кроме метрик, он также учитывает события, связанные с ресурсами, в частности с памятью, такие как вытеснение и `OutOfMemory`.

В нашем примере мы выбрали значение `Off` для параметра `.spec.updatePolicy.updateMode`, но есть еще два значения, каждое из которых определяет свой уровень потенциальной дезорганизации работы масштабируемых подов. Давайте посмотрим, как работают разные значения

updateMode, перечислив их в порядке увеличения разрушительного влияния:

updateMode : Off

Механизм рекомендаций VPA собирает метрики и события подов, а затем выдает рекомендации. Рекомендации всегда хранятся в разделе status ресурса VPA. Однако дальше этого механизм в режиме Off не идет. Он анализирует имеющуюся информацию и выдает рекомендации, но не применяет их. Этот режим можно использовать для получения информации о потреблении ресурсов подами без внесения каких-либо изменений и нарушения их работы. Принятие решения о применении рекомендаций оставлено на усмотрение пользователя.

updateMode : Initial

В этом режиме VPA делает еще шаг вперед. Кроме действий, выполняемых механизмом рекомендацией, дополнительно в работу вовлекается плагин согласования (admission plugin), который применяет рекомендации только ко вновь созданным подам. Например, если масштабирование пода осуществляется вручную, через механизм HPA, при изменении параметров развертывания Deployment или в случае остановки и повторного запуска пода по какой-либо причине, контроллер согласования VPA обновит значение запроса на ресурс.

Этот контроллер является *изменяющим плагином согласования* и переопределяет значения в поле requests новых подов, соответствующих селектору меток VPA. Данный режим не вызывает перезапуск действующих подов

и является частично разрушительным, потому что изменяет требования к ресурсам для вновь создаваемых подов. Это может повлиять на выбор места для запуска нового пода. Более того, может так получиться, что после применения рекомендуемых требований к ресурсам под будет запланирован на другом узле, что может иметь неожиданные последствия. Или, что еще хуже, планировщик не сможет подобрать для пода подходящий узел, если ни на одном узле в кластере не окажется достаточного объема ресурсов.

updateMode : Auto

Кроме создания рекомендаций и их применения к вновь создаваемым подам, как описано выше, в этом режиме VPA вовлекает в работу свой компонент, выполняющий обновления. Этот компонент останавливает запущенные поды, которые соответствуют селектору меток, и запускает их вновь с помощью плагина согласования VPA, который обновляет требования к ресурсам. То есть этот режим является наиболее разрушительным, поскольку принудительно перезапускает все поды для применения рекомендаций, что может вызвать неожиданные проблемы с планированием, как описано выше.

Фреймворк Kubernetes предназначен для управления неизменяемыми контейнерами с неизменяемыми определениями `срес` в подах, как показано на рис. 24.2. Это упрощает горизонтальное масштабирование, но создает проблемы для вертикального масштабирования из-за необходимости останавливать и повторно запускать поды, что может повлиять на процесс планирования и вызвать сбои в

работе. Это верно, даже когда под сокращает требуемый объем ресурсов и хочет освободить уже выделенные ресурсы.

Другая проблема связана с сосуществованием VPA и HPA. В настоящее время эти два механизма действуют независимо друг от друга, что может привести к нежелательному поведению. Например, если HPA использует метрики ресурсов, такие как процессорное время и объем памяти, и VPA влияет на эти же значения, это может привести к одновременному горизонтальному и вертикальному масштабированию подов (то есть произойдет двойное масштабирование).

Мы не будем еще дальше углубляться в детали, потому что механизм VPA все еще находится в состоянии бета-версии и его поведение может измениться. Но имейте в виду, что его применение может значительно улучшить потребление ресурсов.

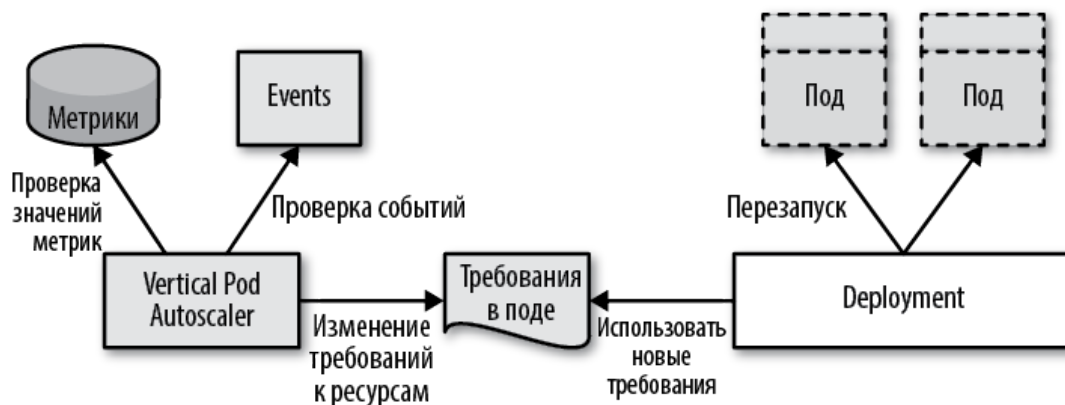


Рис. 24.2. Механизм автоматического вертикального масштабирования подов

Автоматическое масштабирование кластера

Паттерны, представленные в этой книге, используют в основном примитивы и ресурсы, предназначенные для разработчиков, использующих уже настроенный кластер Kubernetes. Поскольку эта тема связана с масштабированием

рабочих нагрузок, в этом разделе мы познакомимся с механизмом автоматического масштабирования кластера Kubernetes Cluster Autoscaler (CA).

Одним из принципов облачных вычислений является оплата фактически потребленных ресурсов. Мы можем использовать облачные услуги, когда они необходимы, и ровно столько, сколько необходимо. Механизм автоматического масштабирования кластера может взаимодействовать с облачными провайдерами, где работает Kubernetes, и запрашивать выделение дополнительных узлов в периоды пиковых нагрузок и освобождать незанятые узлы в другие периоды, снижая затраты на инфраструктуру. Если HPA и VPA выполняют масштабирование на уровне подов и обеспечивают эластичное распределение ресурсов для служб внутри кластера, то CA обеспечивает масштабируемость на уровне узлов и эластичное распределение ресурсов для кластера в облачной инфраструктуре.

CA — это расширение для Kubernetes, которое должно быть включено и настроено минимальным и максимальным количеством узлов. Оно может функционировать, только если кластер Kubernetes работает в облачных инфраструктурах, где узлы могут выделяться и выводиться из эксплуатации по требованию и имеется поддержка Kubernetes CA, таких как AWS, Microsoft Azure и Google Compute Engine.

Cluster API

Все крупные облачные провайдеры поддерживают Kubernetes CA. Однако плагины, реализующие эту поддержку, были написаны самими провайдерами, что вызывает привязку к одному провайдеру и несовместимость в поддержке CA. К счастью, появился проект Cluster API Kubernetes, целью

которого является определение API для создания, настройки и управления кластером. Все крупные провайдеры облачных вычислений, такие как AWS, Azure, GCE, vSphere и OpenStack, поддержали эту инициативу. Он также предусматривает автоматическое масштабирование локально установленных экземпляров Kubernetes. Сердцем Cluster API является контроллер машины, действующий в фоновом режиме, для которого уже существует несколько независимых реализаций, таких как Kubermatic machine-controller или Open-Shift machine-api-operator. Проект Cluster API стоит того, чтобы внимательно следить за его развитием, так как в будущем он может стать основой для любого другого механизма автоматического масштабирования кластера.

CA выполняет две основные операции: добавление новых узлов в кластер и удаление узлов из кластера. Давайте посмотрим, как выполняются эти действия:

Добавление нового узла (масштабирование вверх)

Приложениям с переменной нагрузкой (с пиковыми периодами в течение дня, в выходные или праздничные дни и намного меньшей нагрузкой в другое время) необходима возможность увеличивать объем доступных ресурсов, когда это требуется. Можно, конечно, купить фиксированный объем ресурсов у облачного провайдера, чтобы покрыть потребности в пиковые периоды, но тогда высокая оплата за этот объем в периоды с небольшой нагрузкой уменьшит преимущества облачных вычислений. В таких случаях

автоматическое масштабирование кластера становится по-настоящему полезным.

При горизонтальном или вертикальном масштабировании пода, вручную или посредством HPA или VPA, реплики должны размещаться на узлах, удовлетворяющих требованиям к процессору и памяти. Если в кластере не окажется узла с объемом ресурсов, удовлетворяющим требованиям пода, тогда этот под будет помечен как *не подлежащий планированию* и останется в состоянии ожидания, пока такой узел не будет найден. Механизм SA следит за такими подами, чтобы понять, когда следует добавить новый узел, удовлетворяющий их потребности. Обнаружив такой под, механизм SA изменит размер кластера согласно потребностям.

SA не может добавить в кластер случайный узел — он должен выбрать узел из доступных групп узлов, на которых работает кластер. Предполагается, что все машины в группе узлов имеют одинаковую емкость и одинаковые метки и на них выполняются одинаковые поды, указанные в локальных файлах манифеста или в наборах DaemonSet. Это предположение необходимо, чтобы SA мог оценить, какую дополнительную емкость добавит новый узел в кластер.

Если потребностям ожидающих подов соответствует несколько групп узлов, тогда SA можно настроить для выбора группы узлов с помощью разных стратегий, называемых *механизмами расширения*. Механизм расширения может расширить группу дополнительным узлом, используя критерий минимизации стоимости, минимизации избыточных ресурсов, максимизации числа подов, которые можно разместить на узле или просто случайным образом. Когда выбор будет сделан, облачный провайдер должен за несколько минут подготовить новую

машину и зарегистрировать ее в API Server как новый узел Kubernetes, готовый для размещения ожидающих подов.

Удаление узла (масштабирование вниз)

Масштабирование вниз подов или узлов без прерывания обслуживания всегда сложнее и требует множества проверок. SA выполняет масштабирование вниз, когда нет необходимости выполнять масштабирование вверх, а узел идентифицируется как незанятый. Узел может быть удален, если выполняются следующие основные условия:

- Более половины его емкости не используется, то есть сумма всех требований к процессорному времени и памяти всех подов на этом узле составляет менее 50% от емкости его ресурсов.
- Все поды на узле, доступные для перемещения (которые не запускаются локально с помощью файлов манифеста или наборов DaemonSet), можно переместить на другие узлы. Чтобы убедиться в этом, SA имитирует процесс планирования и определяет будущее местоположение для каждого пода после его остановки на этом узле. Окончательное расположение подов по-прежнему определяется планировщиком и может отличаться, но успех моделирования гарантирует наличие резервных мощностей в кластере.
- Нет никаких других причин, препятствующих удалению узла, как, например, исключение узла из списка доступных для масштабирования вниз посредством аннотаций.

- На узле нет подов, которые нельзя переместить, таких как поды с требованиями PodDisruptionBudget, которые нельзя удовлетворить, поды с локальным хранилищем, поды с аннотациями, предотвращающими вытеснение, поды, созданные без контроллера, или системные поды.

Все эти проверки выполняются с целью гарантировать, что не будет остановлен ни один под, который нельзя запустить на другом узле. Если все предыдущие условия выполняются в течение некоторого времени (по умолчанию 10 минут), узел может быть удален. Для удаления узел помечается как непригодный для планирования и все поды, выполняющиеся на нем, перемещаются на другие узлы.

Схема на рис. 24.3 показывает, как СА взаимодействует с облачным провайдером и Kubernetes в процессе масштабирования кластера.

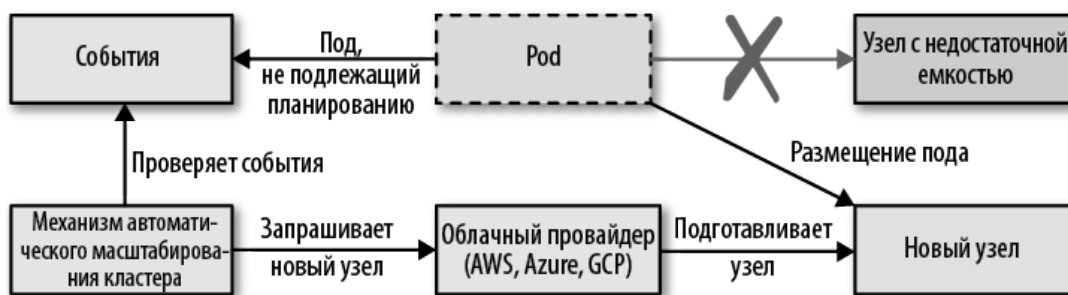


Рис. 24.3. Механизм автоматического масштабирования кластера

Как вы, наверное, уже поняли, масштабирование подов и масштабирование узлов не связаны между собой, а выполняются независимо. НРА или VPA могут анализировать метрики, события и масштабировать поды. Если в какой-то момент емкости кластера оказывается недостаточно, в работу вступает СА и увеличивает ее. Также механизм СА может пригодиться, когда имеются существенные колебания нагрузки на кластер из-за выполнения пакетных заданий, повторяющихся задач, тестов непрерывной интеграции или

других пиковых нагрузок, требующих временно увеличить емкость. Он может увеличить и уменьшить емкость кластера и значительно сэкономить затраты на облачную инфраструктуру.

Уровни масштабирования

В этой главе мы рассмотрели различные способы масштабирования развернутых рабочих нагрузок для удовлетворения меняющихся потребностей в ресурсах. Большинство из перечисленных здесь действий вполне может выполнить оператор-человек, но это не соответствует облачному мышлению. Для крупномасштабного управления распределенными системами необходима автоматизация повторяющихся действий. Лучше автоматизировать масштабирование и позволить операторам-людям сосредоточиться на задачах, которые пока нельзя автоматизировать с помощью *операторов* Kubernetes.

Давайте еще раз окинем взглядом приемы масштабирования, в порядке от более тонких к более грубым, как показано на рис. 24.4.



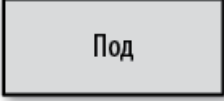
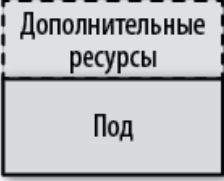
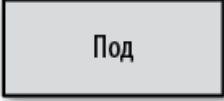
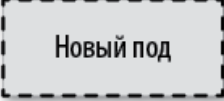
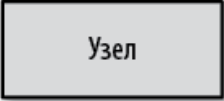
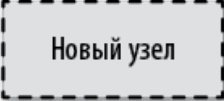
Прием	Действие	Пример масштабирования	
Настройка приложений	Настройка процесса (число потоков выполнения, объем кучи и т. д.)		
Автоматическое вертикальное масштабирование подов	Увеличение/уменьшение ресурсов контейнера		
Автоматическое горизонтальное масштабирование подов	Запуск/остановка подов		
Автоматическое масштабирование кластера	Добавление/удаление узлов		

Рис. 24.4. Уровни масштабирования приложений

Настройка приложений

На самом детальном уровне находится метод, заключающийся в настройке приложений, который мы не рассматривали в этой главе, поскольку он не связан с Kubernetes. Тем не менее самое первое, что можно предпринять, — настроить приложение, выполняющееся в контейнере, чтобы оно оптимально использовало выделенные ресурсы. Это действие не требуется выполнять при каждом масштабировании службы, оно должно быть выполнено изначально, перед выпуском приложения в эксплуатацию. Например, для среды выполнения Java можно настроить размер пула потоков выполнения, обеспечивающий оптимальное использование ресурсов процессорного времени, которые получает контейнер. Также можно настроить объемы различных областей памяти, например размеры кучи, обычной памяти и памяти для стека потоков выполнения.

Корректировка этих значений обычно выполняется с помощью изменений конфигурации, а не кода.

В контейнерных приложениях используются запускающие сценарии, которые могут вычислять оптимальные значения по умолчанию для количества потоков и размеров областей памяти приложения, опираясь на ресурсы, выделенные контейнеру, а не на общую емкость узла. Использование таких сценариев может стать отличным первым шагом. Можно пойти еще дальше и использовать методы и библиотеки, такие как Netflix Adaptive Concurrency Limits, с помощью которых приложение может динамически вычислять свои ограничения путем самопрофилирования и адаптироваться к ним. Это — своеобразное автоматическое масштабирование на уровне приложения, которое устраняет необходимость настройки служб вручную.

Настройка приложений может вызвать регрессию, подобно изменениям в коде, и должна сопровождаться тестированием. Например, изменение размера кучи приложения может привести к его краху из-за ошибки `OutOfMemory`, и горизонтальное масштабирование в этом случае не поможет. С другой стороны, масштабирование подов по вертикали или по горизонтали или выделение большего количества узлов не даст ожидаемого эффекта, если приложение не использует должным образом ресурсы, выделенные для контейнера. То есть настройка на этом уровне масштабирования может повлиять на все другие методы и оказаться разрушительной, но ее необходимо выполнить хотя бы один раз, чтобы добиться оптимального поведения приложения.

Автоматическое вертикальное масштабирование подов

В предположении, что приложение эффективно использует ресурсы контейнера, следующим шагом является установка

правильных запросов и лимитов ресурсов в контейнерах. Выше мы видели, как VPA может автоматизировать процесс определения и применения оптимальных значений, основанных на реальном потреблении. Существенная проблема в данном случае заключается в том, что Kubernetes требует остановки подов и их повторного запуска, что оставляет возможность для коротких или неожиданных периодов прерывания обслуживания. Выделение большего количества ресурсов для контейнера, испытывающего недостаток ресурсов, может сделать под не подлежащим планированию и еще больше увеличить нагрузку на другие экземпляры. Увеличение ресурсов контейнера также может потребовать дополнительной настройки приложения, чтобы обеспечить оптимальное их использование.

Автоматическое горизонтальное масштабирование подов

Предыдущие два метода являются формой вертикального масштабирования; применяя их, мы надеемся улучшить производительность существующих подов путем их настройки, но без изменения их количества. Следующие два метода являются формой горизонтального масштабирования: в этом случае мы не касаемся определений, но меняем количество подов и узлов. Такой подход снижает вероятность появления любых регрессий и сбоев и упрощает автоматизацию. В настоящее время HPA является наиболее популярной формой масштабирования. Первоначально он предлагал лишь минимальные возможности, опираясь только на метрики потребления процессора и памяти, но ныне появилась возможность использовать нестандартные и внешние метрики, помогающие организовать более сложные схемы масштабирования.

Если предположить, что вы выполнили два предыдущих метода и определили правильные значения настроек для самого приложения и потребности в ресурсах контейнера, после этого можно включить НРА и обеспечить адаптацию приложения к изменениям потребностей в ресурсах.

Автоматическое масштабирование кластера

Методы масштабирования с использованием НРА и VPA обеспечивают эластичность только в пределах емкости кластера. Их можно применять, только если кластер Kubernetes обладает достаточной емкостью. Автоматическое масштабирование кластера обеспечивает гибкость на уровне емкости кластера. Механизм СА дополняет другие методы масштабирования, но действует независимо. Он не заботится о причинах появления дополнительного спроса на ресурсы или о том, почему появилась неиспользуемая емкость, является ли это следствием действий человека-оператора или механизма автоматического масштабирования, меняющих профили рабочей нагрузки. Он может расширить кластер, чтобы добавить дополнительную емкость, или сократить его, чтобы сэкономить на затратах на неиспользуемые ресурсы.

Пояснение

Эластичность и различные методы масштабирования — это область Kubernetes, которая продолжает активно развиваться. Поддержка метрик лишь недавно была добавлена в НРА, а VPA все еще находится на стадии экспериментальной разработки. Кроме того, благодаря популяризации модели бессерверных вычислений наибольшим спросом стали пользоваться возможности масштабирования до нуля и быстрого масштабирования вверх. Knative serving — это расширение для

Kubernetes, которое как раз решает эту проблему, обеспечивая основу для масштабирования до нуля, как кратко описывается во врезке «Knative Serving» выше и во врезке «Knative Build» в главе 25. Проект Knative и лежащие в его основе сетки служб продолжают быстро развиваться и представляют новые и очень интересные облачные примитивы. Мы внимательно следим за этой областью облачных вычислений и рекомендуем вам присмотреться к Knative.

Опираясь на описание желаемого состояния распределенной системы, Kubernetes может создавать и поддерживать его. Он также обеспечивает надежность и устойчивость к сбоям, постоянно выполняя мониторинг и автоматически восстанавливая и обеспечивая соответствие текущего состояния желаемому. Гибкость и надежность системы находятся на достаточно высоком уровне для многих нынешних приложений, но Kubernetes не останавливается на достигнутом. Небольшая, но правильно настроенная система Kubernetes сможет надежно функционировать даже при большой нагрузке, масштабируя поды и узлы. То есть под влиянием внешних факторов такая система будет становиться больше и сильнее, пользуясь мощными возможностями Kubernetes.

Дополнительная информация

- Пример эластичного масштабирования (<http://bit.ly/2HwQa6V>).
- Правильный выбор размеров пода для использования механизма автоматического вертикального масштабирования подов (<http://bit.ly/2WlnN9l>).

- Основы автоматического масштабирования в Kubernetes (<http://bit.ly/2U0XoGa>).
- Автоматическое горизонтальное масштабирование подов (<http://bit.ly/2r08Row>).
- Алгоритм HPA (<http://bit.ly/2Fh35Xb>).
- Обзор механизма автоматического горизонтального масштабирования подов (<http://bit.ly/2FlUSRH>).
- Kubernetes Metrics API и клиенты (<https://github.com/kubernetes/metrics/>).
- Автоматическое вертикальное масштабирование подов (<http://bit.ly/2Fixzbn>).
- Настройка автоматического вертикального масштабирования подов (<http://bit.ly/2Hyl0eb>).
- Предложение по автоматическому вертикальному масштабированию подов (<http://bit.ly/2OfAOnW>).
- Репозиторий GitHub с реализацией механизма автоматического вертикального масштабирования подов (<http://bit.ly/2BDnAMZ>).
- Автоматическое масштабирование кластеров в Kubernetes (<http://bit.ly/2TkNQL9>).
- Обзор библиотеки Netflix Adaptive Concurrency Limits (<http://bit.ly/2JuXxxx>).
- Часто задаваемые вопросы по автоматическому масштабированию кластеров (<http://bit.ly/2Cum0NH>).

- Cluster API (<http://bit.ly/2D133T9>).
- Kubermatic Machine-Controller (<http://bit.ly/2VeTqae>).
- OpenShift Machine API Operator (<http://bit.ly/2ul7TzP>).
- Knative (<https://cloud.google.com/knative/>).
- Knative: организация бессерверных служб (<https://red.ht/2HvenKZ>).
- Учебник по Knative (<http://bit.ly/2HT70x9>).

22 В случае с несколькими подами в качестве *текущегоЗначенияМетрики* используется среднее значение использования процессора.

Глава 25. Построитель образов

Kubernetes — это управляющий механизм общего назначения, который прекрасно подходит не только для запуска приложений, но и для создания образов контейнеров. Паттерн *Image Builder* (Построитель образов) объясняет, почему имеет смысл создавать образы контейнеров в кластере и какие методы создания образов существуют сегодня в Kubernetes.

Задача

Все паттерны, представленные выше в этой книге, касались работы приложений в Kubernetes. Мы узнали, как разрабатывать и подготавливать наши приложения для работы в облачном окружении. Но как *собираются* сами приложения? Классический подход заключается в создании образов контейнеров вне кластера, добавлении их в реестр и использовании ссылок на них в описаниях развертываний Deployment. Однако сборка образов контейнеров внутри кластера имеет несколько преимуществ.

Если экономическая политика компании позволяет иметь только один кластер для всего, тогда сборка и запуск приложений в одном месте могут значительно снизить затраты на обслуживание. Это также упрощает планирование емкости и снижает затраты на потребление ресурсов.

Как правило, для сборки образов используются системы непрерывной интеграции (Continuous Integration, CI), такие как Jenkins. Сборка с помощью CI-системы — это задача планирования, эффективного поиска свободных вычислительных ресурсов для заданий на сборку. В основе

Непривилегированная сборка

Когда сборка выполняется в Kubernetes, кластер полностью контролирует ее процесс, но ему также нужны более высокие стандарты безопасности, поскольку сборка выполняется не изолированно. Для сборки в кластере важно, чтобы она запускалась без привилегий root. К счастью, в настоящее время существует множество способов организовать сборку в так называемом *непривилегированном режиме*, когда не требуется повышенных привилегий.

Проект Docker достиг больших успехов в массовом распространении контейнерных технологий благодаря своим непревзойденным возможностям. Docker основан на архитектуре клиент-сервер с демоном Docker, работающим в фоновом режиме и получающим инструкции от клиента через REST API. Привилегии root нужны этому демону в основном для управления сетью и томами. К сожалению, такой подход создает угрозу безопасности, поскольку злоумышленник может просочиться в контейнеры ненадежных процессов и получить контроль над всем хостом. Эта проблема относится не только к выполнению контейнеров, но и к их сборке, потому что сборка также происходит внутри контейнера, когда демон Docker выполняет произвольные команды.

Со временем было создано множество проектов с целью дать возможность выполнять сборку в Docker без привилегий root и сузить круг возможностей для злоумышленников. Некоторые из них (например, *Jib*) запрещают запускать команды во время сборки, другие используют иные методы. На момент написания

этой книги самыми известными инструментами создания образов без повышенных привилегий были *img*, *buildah* и *Kaniko*. Кроме того, система S2I, описанная в разделе «Из исходного кода в образ» ниже, также выполняет сборку образов без привилегий *root*.

Kubernetes лежит очень сложный планировщик, который идеально подходит для такого рода задач.

При использовании методологии непрерывной доставки (Continuous Delivery, CD), когда требуется организовать переход от *создания образов* к *работающим контейнерам*, если сборка происходит в том же кластере, оба этапа используют одну и ту же инфраструктуру, что упрощает переход. Например, предположим, что в образе, используемом в качестве основы для всех приложений, обнаружена новая уязвимость. Когда ваша команда исправит эту проблему, вам придется пересобрать все образы приложений, которые зависят от этого базового образа, и заменить запущенные приложения новыми образами. В случае, если реализован паттерн *Image Builder* (Построитель образов), кластер будет знать, как собрать образ и как его развернуть, и сможет автоматически выполнить повторное развертывание после изменения базового образа. Ниже, в разделе «Сборка в OpenShift», я расскажу, как такая автоматизация реализуется в OpenShift.

А теперь, выяснив, какие преимущества дает сборка образов внутри платформы, посмотрим, какие существуют методы создания образов в кластере Kubernetes.

Решение

Один из первых и наиболее зрелых способов сборки образов в кластере Kubernetes основан на использовании подсистемы сборки в OpenShift. Она поддерживает несколько вариантов сборки образов. Один из них — *сборка образа из исходных кодов* (Source-to-Image, S2I) — консервативный способ организации сборки с использованием так называемых строителей образов. Мы подробнее рассмотрим S2I и способ сборки образов в OpenShift в разделе «Сборка в OpenShift» ниже.

Другой механизм сборки образов внутри кластера основан на использовании *Knative Build*. Этот инструмент работает поверх Kubernetes и сетки служб Istio и является одной из основных частей Knative (<https://cloud.google.com/knative/>), платформы для создания, развертывания и управления бессерверными приложениями. На момент написания этой книги Knative все еще считался очень молодым проектом и продолжал быстро развиваться. В разделе «Knative Build» я кратко расскажу о проекте Knative и приведу примеры сборки образов в кластере Kubernetes с помощью Knative Build.

Начнем знакомство с механизмами сборки с OpenShift.

Сборка в OpenShift

Red Hat OpenShift — это корпоративный дистрибутив Kubernetes. Кроме поддержки всего того, что поддерживает Kubernetes, он добавляет несколько дополнительных функций, востребованных на предприятиях, таких как интегрированный реестр образов контейнеров, поддержка единой службы авторизации и новый пользовательский интерфейс, а также добавляет в Kubernetes возможность создания собственных образов. Для сообщества открытого программного обеспечения выпускается дистрибутив OKD (<https://www.okd.io/>) — ранее известный как OpenShift Origin, — поддерживающий все возможности OpenShift.

Дистрибутив OpenShift стал первым предложившим интегрированную в кластер возможность сборки образов под управлением Kubernetes. Он поддерживает несколько стратегий создания образов:

Из исходного кода в образ (Source-to-Image, S2I)

Принимает исходный код приложения и создает выполняемый артефакт с помощью построителя образа S2I для конкретного языка, а затем помещает образы в интегрированный реестр.

Сборка средствами Docker

Использует *Dockerfile* плюс каталог контекста и создает образ, как это сделал бы демон Docker.

Конвейерная сборка

Представляет процесс сборки в виде серии заданий, выполняющихся под управлением сервера Jenkins, позволяя пользователю настроить конвейер Jenkins на свой лад.

Нестандартная сборка

Дает полный контроль над процессом сборки образов. В этом случае вы должны сами создать образ в контейнере для сборки и отправить его в реестр.

Данные для сборки могут извлекаться из разных источников.

Git

Репозиторий, откуда извлекаются исходные данные, определяется в виде URL.

Dockerfile

Файл *Dockerfile*, который доступен непосредственно, как часть ресурса конфигурации сборки.

Образ

Образ другого контейнера, из которого извлекаются файлы для текущей сборки. Поддержка этого типа источников дает возможность выполнять *конвейерную сборку*, как будет показано в листинге 25.2.

Ресурс Secret

Этот ресурс можно использовать для передачи конфиденциальной информации в процесс сборки.

Двоичные данные

Источник этого типа предназначен для передачи данных извне, которые должны быть подготовлены перед началом сборки.

Выбор того или иного типа источника зависит от стратегии сборки. *Двоичные данные* и *Git* являются взаимоисключающими типами источников. Источники всех остальных типов можно объединять или использовать по отдельности. Ниже, в листинге 25.1, я покажу, как это делается.

Вся информация о сборке определяется в центральном объекте ресурса BuildConfig. Этот ресурс можно создать

напрямую, применив его к кластеру, или с помощью клиента командной строки `oc`, который является эквивалентом `kubectl` в OpenShift. Клиент `oc` поддерживает команду `build`, с помощью которой можно определить настройки сборки и запустить ее.

Прежде чем взглянуть на `BuildConfig`, мы должны разобраться с двумя дополнительными понятиями, характерными для OpenShift.

ImageStream — это ресурс OpenShift, который ссылается на один или несколько образов контейнеров. Он имеет некоторое сходство с репозиторием Docker, который также содержит несколько образов с разными тегами. OpenShift отображает фактический образ с тегом в ресурс `ImageStreamTag` так, что `ImageStream` (репозиторий) получает список ссылок на ресурсы `ImageStreamTag` (образы с тегами). Зачем нужна эта дополнительная абстракция? Она позволяет OpenShift генерировать события для `ImageStreamTag` при обновлении образа в реестре. Образы создаются во время сборки или когда образ помещается во внутренний реестр OpenShift. При таком подходе механизм сборки или развертывания получает возможность прослушивать эти события и запускать новую сборку или развертывание.



Для подключения `ImageStream` к развертыванию вместо Kubernetes-ресурса `Deployment` OpenShift использует ресурс `DeploymentConfig`, который может содержать только ссылки на образы контейнеров. Тем не менее в OpenShift все равно можно использовать ресурсы `Deployment`, если не планируется использовать ресурсы `ImageStream`.

Другое понятие — *триггер*, который можно рассматривать как обработчик событий. Одним из возможных триггеров является `imageChange`, который реагирует на события, публикуемые при изменении `ImageStreamTag`. В качестве реакции такой триггер может, например, запустить повторную сборку другого образа или повторное развертывание подов, основанных на этом образе. Узнать больше о триггерах и их разновидностях можно в документации OpenShift (<https://red.ht/2FrDIDj>).

Из исходного кода в образ

Давайте кратко рассмотрим, как выглядит построитель образов S2I. Не вдаваясь в детали, отметим, что построитель образов S2I — это стандартный образ контейнера с набором сценариев S2I, которому мы должны предоставить две обязательные команды:

assemble

Сценарий, который вызывается в момент запуска сборки. Его задача — получить исходный код из источников, определяемых настройками, скомпилировать, если необходимо, и скопировать получившиеся артефакты в соответствующие местоположения.

run

Используется как точка входа для этого образа. OpenShift вызывает этот сценарий при развертывании образа. Сценарий `run` использует сгенерированные артефакты для запуска приложения.

При желании вы также можете создать сценарий для вывода сообщения о порядке использования, сохраняющий сгенерированные артефакты для так называемых *инкрементальных сборок*, которые будут доступны сценарию `assemble` в последующих запусках сборки, или добавляющий некоторые проверки.

Рассмотрим поближе механизм сборки S2I, изображенный на рис. 25.1. Он состоит из двух компонентов: построителя образов и средства ввода исходного кода. Оба объединяются системой сборки S2I в момент, когда происходит запуск сборки — либо триггером, после получения события, либо вручную. Когда сборка образа завершится, например, путем компиляции исходного кода, контейнер копируется в образ и передается в сконфигурированный `ImageStreamTag`. Этот образ содержит скомпилированные и подготовленные артефакты и сценарий `run` в качестве точки входа.

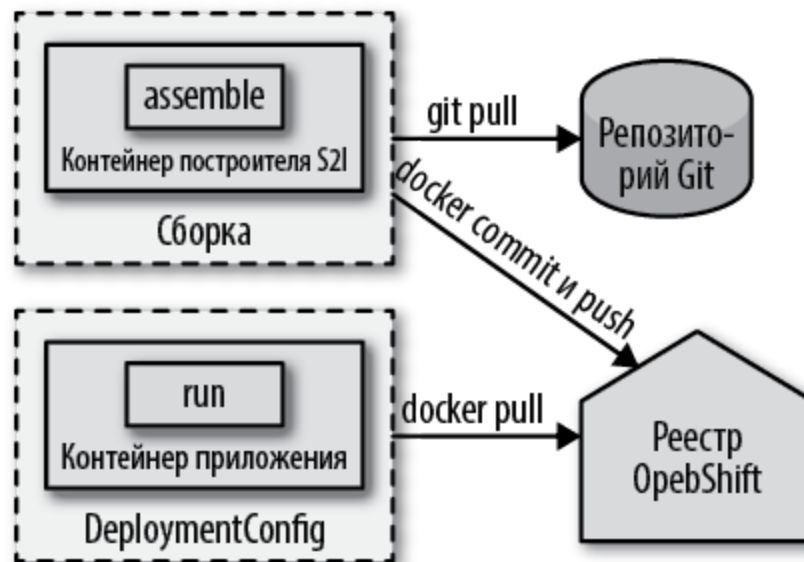


Рис. 25.1. Система сборки S2I с источником входных данных Git

В листинге 25.1 показано простое описание сборки с использованием образа построителя Java. Это описание

определяет процесс сборки, который получает исходный код и образ построителя, создает выходной образ и передает его в ImageStreamTag. Он может запускаться вручную, командой `oc start-build`, или автоматически, при изменении образа построителя.

Листинг 25.1. Сборка S2I с использованием образа построителя Java

```
apiVersion: v1
kind: BuildConfig
metadata:
  name: random-generator-build
spec:
  source: ❶
    git:
      uri:
        https://github.com/k8spatterns/random-generator
  strategy: ❷
    sourceStrategy:
      from:
        kind: DockerImage
        name: fabric8/s2i-java
  output: ❸
    to:
      kind: ImageStreamTag
      name: random-generator-build:latest
  triggers: ❹
  - type: ImageChange
```

❶ Ссылка на исходный код, который требуется извлечь; в данном случае он извлекается из репозитория на GitHub.

❷ `sourceStrategy` включает режим S2I, и образ построителя извлекается непосредственно из Docker Hub.

③ Определяется `ImageStreamTag` для передачи сгенерированного образа. Передача выполняется контейнером построителя после выполнения сценария `assemble`.

④ Настройка автоматического запуска сборки при изменении образа построителя.

S2I — это надежный механизм для создания образов приложений. Он безопаснее простых механизмов сборки Docker, поскольку процесс сборки протекает под полным контролем проверенных образов построителей. Тем не менее этот подход имеет некоторые недостатки.

S2I может работать очень медленно, особенно когда собирается сложное приложение со множеством зависимостей. В отсутствие какой-либо оптимизации механизм S2I заново загружает все зависимости для каждой сборки. Если Java-приложение собирается с помощью Maven, кэширование не выполняется, как при локальной сборке. Чтобы не загружать зависимости из интернета снова и снова, рекомендуется настроить в кластере внутренний репозиторий Maven, который служит кэшем, а также настроить образ построителя, чтобы он обращался к этому внутреннему репозиторию, а не загружал артефакты из удаленных репозиториев.

Другой способ уменьшить время сборки — использовать *инкрементальные сборки* с S2I, позволяющие повторно использовать артефакты, созданные или загруженные в предыдущей сборке S2I. Однако из-за того, что большая часть данных копируется из ранее созданного образа в текущий собираемый контейнер, выигрыш в производительности получается небольшим по сравнению с использованием локального кэша в кластере, где хранятся зависимости.

Другой недостаток S2I заключается в том, что сгенерированный образ содержит также все окружение сборки. Это не только увеличивает размер образа, но и расширяет круг

возможностей для потенциальной атаки, поскольку инструменты разработчика также могут иметь уязвимости.

Чтобы избавиться от ненужных инструментов сборки, таких как Maven, OpenShift предлагает использовать прием *конвейерной сборки*, когда из результатов сборки S2I создается уменьшенный образ со средой выполнения. Подробнее этот прием мы рассмотрим в разделе «Конвейерная сборка» ниже.

Сборка средствами Docker

OpenShift поддерживает также сборку средствами Docker. Такая сборка выполняется путем монтирования сокета демона Docker непосредственно в сборочный контейнер, который затем используется командой `docker build`. Источником данных для сборки средствами Docker является *Dockerfile* и каталог контекста. Также можно использовать источник Image, ссылающийся на произвольный образ, откуда извлекаются файлы и копируются в каталог контекста. Как уже говорилось, этот метод вместе с триггерами можно использовать для конвейерной сборки.

Кроме того, есть возможность использовать стандартный многоступенчатый *Dockerfile* для разделения этапов сборки и среды выполнения. В нашей репозитории с примерами (<http://bit.ly/2CxnnuX>) вы найдете действующий пример такой многоступенчатой сборки средствами Docker, создающий точно такой же образ, что и конвейерная сборка, описанная в следующем разделе.

Конвейерная сборка

Механика конвейерной сборки показана на рис. 25.2. Процесс конвейерной сборки включает этап начальной сборки S2I, на котором создается выполняемый артефакт, например

двоичный выполняемый файл. Затем, на втором этапе, обычно выполняемом средствами Docker, этот артефакт извлекается из сгенерированного образа.

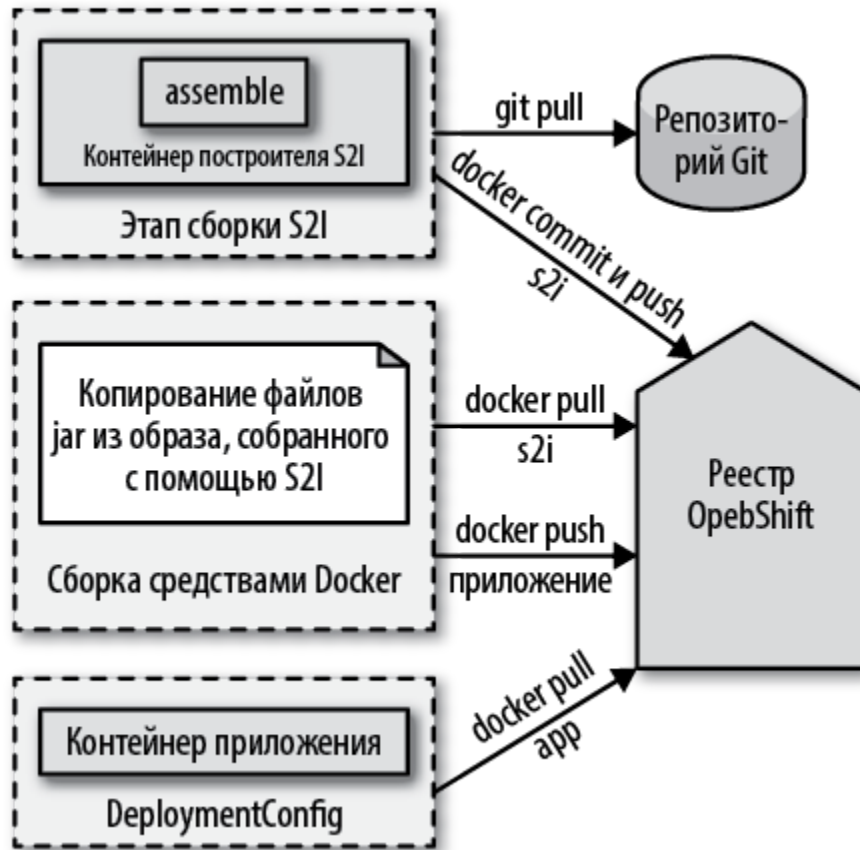


Рис. 25.2. Конвейерная сборка с использованием S2I для компиляции и Docker для сборки образа приложения

В листинге 25.2 показана конфигурация второго этапа сборки, который использует файл JAR, сгенерированный в листинге 25.1. Образ, который в конечном итоге передается в ImageStream `random-generator-runtime`, можно использовать в DeploymentConfig для запуска приложения.



Обратите внимание, что в листинге 25.2 используется триггер, который следит за результатами сборки S2I. Этот триггер автоматически вызывает повторную сборку образа времени выполнения после каждой сборки S2I, поэтому оба ImageStream всегда синхронизированы.

Листинг 25.2. Сборка средствами Docker для создания образа приложения

```
apiVersion: v1
kind: BuildConfig
metadata:
  name: runtime
spec:
  source:
    images:
      - from: ❶
        kind: ImageStreamTag
        name: random-generator-build:latest
    paths:
      - sourcePath: /deployments/.
        destinationDir: "."
    dockerfile: |- ❷
      FROM openjdk:8-alpine
      COPY *.jar /
      CMD java -jar /*.jar
  strategy: ❸
    type: Docker
  output: ❹
```

```
to:
  kind: ImageStreamTag
  name: random-generator:latest
triggers:
  ⑤
- imageChange:
  automatic: true
  from:
    kind: ImageStreamTag
    name: random-generator-build:latest
  type: ImageCh
```

① Источник ссылается на образ в ImageStream, содержащий результаты сборки S2I, и определяет каталог в образе, где находится скомпилированный архив JAR.

② *Dockerfile* для сборки средствами Docker, который копирует архив JAR из ImageStream, сгенерированного сборкой S2I.

③ Стратегия, включающая режим сборки средствами Docker.

④ Повторная сборка запускается автоматически, если результат сборки S2I — ImageStream — изменится и благополучно будет создан новый архив JAR.

⑤ Регистрация приемника событий, связанных с изменением образа, и выполнение повторного развертывания после добавления обновленного образа в ImageStream.

Полный пример с инструкциями по установке можно найти в нашем репозитории (<http://bit.ly/2CxnnuX>).

Как уже упоминалось, сборка в OpenShift, включая наиболее известный режим S2I, является одним из ранних и наиболее зрелых способов безопасной сборки образов контейнеров в кластере Kubernetes.

Теперь рассмотрим другой способ сборки образов контейнеров в кластере Kubernetes без всяких расширений.

Knative Build

В 2018 году компания Google запустила проект Knative с целью привнести в Kubernetes дополнительные возможности, имеющие прямое отношение к приложениям.

Основой для Knative стала *сетка служб Istio* (<https://istio.io/>), которая предлагает готовые инфраструктурные службы для управления трафиком, наблюдения и поддержания безопасности. Сетки служб используют паттерн *Sidcars* (Прицеп) для передачи приложениям новых инфраструктурных возможностей.

Кроме сетки служб, проект Knative предлагает дополнительные службы, в первую очередь предназначенные для разработчиков приложений:

Knative Serving

Для поддержки масштабирования до нуля приложений, которые могут использоваться, например, на платформах FaaS (Function-as-a-Service — функция как служба). Наряду с паттерном, описанным в главе 24 «Эластичное масштабирование», и поддержкой сетки служб, Knative Serving позволяет также масштабировать от нуля до произвольного числа реплик.

Knative Eventing

Для доставки по каналам событий из источников в приемники. События могут запускать службы Service,

используемые в роли приемников и масштабируемые от нуля.

Knative Build

Для компиляции исходного кода приложений в образах контейнеров внутри кластера Kubernetes. Продолжением этой службы стал проект Tekton Pipelines, который в конечном итоге заменит Knative Build.

Оба проекта, Istio и Knative, реализованы по паттерну *Operator* (Operator) и используют определения нестандартных ресурсов (CustomResourceDefinition, CRD) для объявления своих предметных ресурсов.

В оставшейся части этого раздела мы сфокусируем свое внимание на Knative Build реализации паттерна *Image Builder* (Построитель образов) в Knative.



Проект Knative Build в первую очередь предназначен для разработчиков инструментов, стремящихся предоставить конечному пользователю удобный пользовательский интерфейс и скрыть от него сам процесс сборки. Здесь мы лишь в общих чертах рассмотрим строительные блоки Knative Build. Проект быстро развивается, и в будущем его могут даже заменить последующим проектом, таким как Tekton Pipelines, но основная механика, скорее всего, останется прежней. Однако некоторые детали могут измениться, поэтому обращайтесь к примерам кода

(<http://bit.ly/2CxnnuX>), которые мы постоянно обновляем по мере появления новых версий проектов Knative.

Проект Knative создавался с целью предоставить строительные блоки для интеграции с существующими решениями CI/CD²³. Это не решение CI/CD для создания образов контейнеров. Ожидается, что со временем будет появляться все больше и больше таких решений, но сейчас давайте посмотрим, какие строительные блоки оно реализует.

Простая сборка

Ресурс Build является основой Knative Build. Он определяет конкретные шаги, которые должен выполнить оператор Knative Build. Ниже перечислены основные ингредиенты, использованные в листинге 25.3:

- Параметр `source` определяет местоположение исходного кода приложения. Исходный код может храниться в репозитории Git, как в листинге 25.3, в других удаленных хранилищах, таких как Google Cloud Storage, или даже в произвольном контейнере.
- Параметр `steps` описывает шаги, которые нужно выполнить, чтобы превратить исходный код в образ контейнера. Каждый шаг ссылается на образ построителя, используемый для его выполнения. Каждый шаг имеет доступ к тому, смонтированному в каталог `/workspace`, который содержит исходный код и используется для обмена данными между шагами.

Роль исходного кода в этом примере снова играет наш пример Java-проекта, который размещается в GitHub и собирается с помощью Maven. Образ построителя — это образ контейнера, содержащего Java и Maven. Сборка образа выполняется инструментом Jib (<https://github.com/GoogleContainerTools/jib>) без привлечения демона Docker и отправляется в реестр.

Листинг 25.3. Сборка Java-проекта в Knative с использованием Maven и Jib

```
apiVersion: build.knative.dev/v1alpha1
kind: Build
metadata:
  name: random-generator-build-jib ❶
spec:
  source: ❷
    git:
      url:
        https://github.com/k8spatterns/random-
        generator.git
      revision: master
  steps: ❸
  - name: build-and-push
    image: gcr.io/cloud-builders/mvn ❹
    args: ❺
    - compile
      - com.google.cloud.tools:jib-maven-
plugin:build
    -
    -
  Djib.to.image=registry/k8spatterns/random-
  generator
  workingDir: /workspace ❻
```

❶ Имя объекта-построителя.

② Ссылка на источник исходного кода с URL репозитория GitHub.

③ Этапы сборки.

④ Образ с Java и Maven, который используется на этом этапе сборки.

⑤ Аргументы для передачи в контейнер построителя, который запускает Maven для компиляции, создает и пересылает образ контейнера с помощью `jib-maven-plugin`.

⑥ Каталог `/workspace` — общий для всех этапов сборки.

Также интересно заглянуть внутрь, чтобы узнать, как оператор Knative Build выполняет сборку.

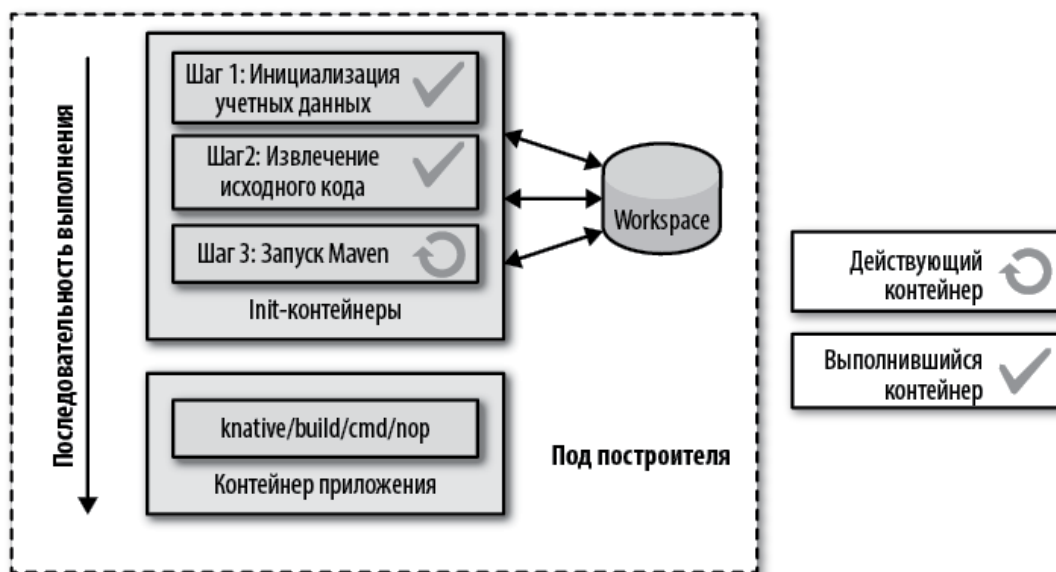


Рис. 25.3. Сборка в Knative с использованием init-контейнеров

На рис. 25.3 показано, как нестандартный ресурс сборки Build превращается в простые ресурсы Kubernetes. Ресурс Build превращается в под, а этапы сборки — в цепочку *init-контейнеров*, которые вызываются один за другим. Init-контейнеры создаются неявно. В нашем примере один из них предназначен для инициализации учетных данных, необходимых для взаимодействий с внешними

репозиториями, второй `init`-контейнер реализует извлечение исходных кодов из GitHub и третий `init`-контейнер просто реализует шаги, осуществляющие сборку. Когда все `init`-контейнеры завершатся, основному контейнеру не останется ничего, как просто завершиться.

Макеты для сборки

Листинг 25.3 определяет только один шаг сборки, но обычно сборка состоит из нескольких шагов. Для определения шагов, многократно используемых в аналогичных сборках, можно использовать ресурс макета сборки `BuildTemplate`.

Один из таких макетов показан в листинге 25.4. Он определяет три шага:

1. Создание JAR-файла с помощью пакета `mvn`.
2. Создание *Dockerfile*, который копирует этот JAR-файл в образ контейнера и запускает его командой `java -jar`.
3. Создание и отправка образа контейнера с помощью построителя `Kaniko`. `Kaniko` — это инструмент, разработанный в компании Google, который предназначен для сборки образов контейнеров из *Dockerfile* внутри контейнера с помощью локального демона `Docker` в пространстве пользователя.

Макет `BuildTemplate` напоминает ресурс `Build`, но, в отличие от последнего, поддерживает параметры, которые заполняются в момент использования макета. В этом примере используется единственный параметр `IMAGE`. С его помощью определяется целевой образ.

Листинг 25.4. Макет сборки `Knative`, использующий `Maven` и `Kaniko`

```

apiVersion: build.knative.dev/v1alpha1
kind: BuildTemplate
metadata:
  name: maven-kaniko
spec:
  parameters:
    - name: IMAGE ❶
      description: The name of the image to
create and push
  steps:
    - name: maven-build ❷
      image: gcr.io/cloud-builders/mvn
      args:
        - package
      workingDir: /workspace
    - name: prepare-docker-context ❸
      image: alpine
      command: [ .... ]
    - name: image-build-and-push ❹
      image: gcr.io/kaniko-project/executor
      args:
        - --context=/workspace
        - --destination=${IMAGE} ❺

```

❶ Список параметров макета.

❷ Этап компиляции и упаковки приложения на Java с помощью Maven.

❸ Этап создания *Dockerfile* для копирования и запуска сгенерированного JAR-файла. Детали реализации здесь опущены, но вы сможете найти их в примерах на сайте GitHub.

④ Этап, вызывающий Kaniko для сборки и отправки контейнера.

⑤ Адрес отправки образа определяется параметром `IMAGE`.

Этот макет можно указать в ресурсе Build вместо списка шагов, как показано в листинге 25.5. Как видите, макету достаточно передать параметр с именем образа контейнера приложения, который требуется создать, и его можно использовать для создания различных приложений.

Листинг 25.5. Определение сборки в Knative Build с использованием макета

```
apiVersion: build.knative.dev/v1alpha1
kind: Build
metadata:
  name: random-generator-build-chained
spec:
  source: ❶
    git:
      url:
        https://github.com/k8spatterns/random-
        generator.git
      revision: master
  template: ❷
    name: maven-kaniko
    arguments:
      - name: IMAGE ❸
        value: registry:80/k8spatterns/random-
        generator
```

- ❶ Определение источника с исходным кодом.
- ❷ Ссылка на макет, объявленный в листинге 25.4.
- ❸ Описание образа для передачи в макет в виде параметра.

Множество predefined паттернов можно найти в репозитории Knative build-templates.

Этот пример завершает наш краткий тур по Knative Build. Как уже упоминалось, этот проект еще очень молодой и детали его реализации еще могут измениться, но основные механизмы, описанные здесь, должны остаться прежними.

Пояснение

Итак, мы познакомились с двумя способами создания образов контейнеров в кластере. Система сборки OpenShift демонстрирует одно из главных преимуществ, которые дает сборка и запуск приложения в одном кластере. С помощью триггеров ImageStream, реализованных в OpenShift, можно не только связывать вместе несколько сборок, но и повторно развертывать приложение, если в процессе сборки обновится образ контейнера вашего приложения. Это особенно удобно, когда за этапом сборки обычно следует этап развертывания. Интеграция сборки и развертывания — это шаг вперед к святому Граалю технологии непрерывного развертывания. Сборка в OpenShift с помощью S2I — проверенная и доказавшая свою надежность технология, но в настоящее время S2I можно использовать только в дистрибутиве OpenShift платформы Kubernetes.

Knative Build — еще одна реализация паттерна *Image Builder* (Построитель образов). Основная цель Knative Build — преобразовать исходный код в готовый образ контейнера и передать его в реестр, чтобы потом его можно было загрузить с помощью развертывания Deployment. Эти шаги выполняются построителями образов, специализированными для поддержки разных технологий. Knative Build не зависит от конкретных

этапов сборки, но управляет жизненным циклом сборки и ее планированием.

Knative Build — это еще довольно молодой проект (по состоянию на 2019 год), который предлагает строительные блоки для сборки образов в кластере. Он не представляет особого интереса для конечного пользователя и в основном ориентирован на разработчиков инструментов. Можно предположить, что новые и существующие инструменты будут поддерживать Knative Build или другие проекты, основанные на нем, поэтому вы наверняка увидите еще немало реализаций паттерна *Image Builder* (Построитель образов).

Дополнительная информация

- Примеры построителей паттернов (<http://bit.ly/2FpCkkL>).
- Jib (<https://github.com/GoogleContainerTools/jib>).
- Img (<https://github.com/genuinetools/img>).
- Buildah (<https://github.com/projectatomic/buildah>).
- Kaniko (<https://github.com/GoogleContainerTools/kaniko>).
- Описание системы сборки в OpenShift (<http://bit.ly/2HILD0E>).
- Многоступенчатый *Dockerfile* (<http://bit.ly/2YfUY63>).
- Конвейерная сборка с S2I (<https://red.ht/2Jqzlw9>).
- Триггеры сборки (<https://red.ht/2FrDIDj>).
- Описание сборки образов из исходных кодов Source-to-Image (<https://github.com/openshift/source-to-image>).

- Инкрементальные сборки с S2I (<https://red.ht/2TSGxp9>).
- Knative (<https://cloud.google.com/knative/>).
- Сборка образов контейнеров в кластере Kubernetes с помощью Knative Build (<http://bit.ly/2YJuZUC>).
- Knative Build (<https://github.com/knative/build>).
- Tekton Pipelines (<https://github.com/knative/build-pipeline>).
- Knative: сборка бессерверной службы (<https://red.ht/2Oew8Pj>).
- Введение в Knctl: простейший способ использования Knative (<https://ibm.co/2Hwnmvw>).
- Интерактивное руководство по Knative Build (<http://bit.ly/2OewLZb>).
- Макеты в Knative Build (<https://github.com/knative/build-templates>).

[23](#) Continuous integration/continuous delivery — непрерывная интеграция и развертывание (доставка). — *Примеч. пер.*

Послесловие

Универсальная платформа

В настоящее время Kubernetes считается самой популярной платформой управления контейнерами. Она разрабатывается и поддерживается в тесном сотрудничестве всеми основными компаниями — разработчиками программного обеспечения и предлагается в качестве услуги всеми основными поставщиками облачных услуг. Она поддерживает системы Linux и Windows и все основные языки программирования. Kubernetes может управлять приложениями без состояния и с состоянием, пакетными заданиями, периодическими заданиями и бессерверными рабочими нагрузками. Это новый уровень организации переносимых приложений и общий знаменатель для облачных вычислений в целом. Если вы разработчик ПО, которое используется в облачной среде, тогда Kubernetes почти неизбежно станет частью вашей повседневной жизни.

Многообразие обязанностей

В последние годы приложения предъявляют облачным платформам все больше и больше нефункциональных требований. Например, Kubernetes принимает на себя такие обязанности, как подготовка, развертывание, обнаружение служб, управление конфигурациями, управление заданиями, изоляция ресурсов и проверка работоспособности. С ростом популярности архитектуры микросервисов реализация даже простой службы требует хорошего понимания технологии распределенных вычислений и основ управления

контейнерами. Как следствие, разработчик должен свободно владеть современным языком программирования для реализации бизнес-функций и не менее свободно владеть облачными технологиями для удовлетворения нефункциональных требований.

О чем мы говорили

В этой книге мы рассмотрели 24 самых популярных паттерна использования Kubernetes, сгруппированных следующим образом:

- Часть I *Основные паттерны* представляет принципы, которым должны соответствовать контейнерные приложения в облачном окружении. Вы должны стремиться следовать этим рекомендациям, независимо от природы приложения и возможных ограничений. Это поможет гарантировать пригодность ваших приложений для автоматизированного управления ими в Kubernetes.
- Часть II *Поведенческие паттерны* описывает механизмы взаимодействий между подами и управляющей платформой. В зависимости от типа рабочей нагрузки под может выполняться до завершения, как пакетное задание, или запускаться периодически. Он может работать как фоновая служба или как служба-одиночка (синглтон). Выбор правильного примитива управления поможет вам обеспечить желаемые гарантии для запуска пода.
- Часть III *Структурные паттерны* основное внимание уделяет структурированию и организации контейнеров в поды для разных вариантов использования. Создание облачных контейнеров — это только первый шаг, но его недостаточно.

Следующий шаг — повторное использование контейнеров и их объединение в поды для достижения желаемого результата.

- Часть IV *Конфигурационные паттерны* охватывает приемы настройки и адаптации приложений в облаке. Каждое приложение должно быть настроено, но нет универсального способа сделать это, который годился бы для всех приложений. В этой части мы исследовали паттерны от самых распространенных до самых специализированных.
- Часть V *Дополнительные паттерны* исследует более сложные темы, которые не вписываются ни в одну из других категорий. Некоторые паттерны, такие как *Controller* (Контроллер), являются достаточно зрелыми — в частности, на паттерне Контроллер построена сама платформа Kubernetes, — а некоторые все еще являются новыми и могут измениться к тому времени, когда вы будете читать эту книгу. Но эти паттерны основаны на базовых идеях, знать которые должны все разработчики, использующие облачные технологии.

В заключение

Все хорошее когда-нибудь заканчивается, подошла к концу и эта книга. Надеемся, что она вам понравилась и изменила ваше представление о Kubernetes. Мы искренне верим, что платформа Kubernetes и заложенные в ней идеи станут такими же фундаментальными, как идеи объектно-ориентированного программирования. Эта книга — наша попытка повторить труд «Банды четырех», но уже применительно к управлению контейнерами. Мы надеемся, что это не конец, а начало вашего путешествия в Kubernetes.

Удачной работы с kubect !!

Об авторах

Билджин Ибрам (Bilgin Ibryam, @bibryam) — главный архитектор в Red Hat, член Apache Software Foundation и участник нескольких проектов с открытым исходным кодом. Блогер и популяризатор программного обеспечения с открытым исходным кодом, энтузиаст блокчейна, лектор и автор книги «Camel Design Patterns». Имеет более чем десятилетний опыт создания и проектирования высокомасштабируемых, отказоустойчивых распределенных систем.

Билджин занимается наставничеством, программированием и оказанием помощи компаниям в создании успешных решений с открытым исходным кодом. В настоящее время основной его работой является интеграция приложений, организация корпоративных блокчейнов, проектирование распределенных систем, а также разработка микросервисов и облачных приложений в целом.

Доктор Роланд Хасс (Roland Huß, @ro14nd) — главный разработчик в Red Hat, который осуществляет техническое руководство проектом Fuse Online и недавно вошел в группу разработки бессерверных вычислений для работы над Knative. Уже больше 20 лет занимается разработкой на Java и недавно обрел вторую любовь в лице языка Golang. Однако он никогда не забывал свое прошлое системного администратора. Роланд принимает активное участие в проектах с открытым исходным кодом, является ведущим разработчиком моста JMX-HTTP Jolokia и некоторых популярных инструментов сборки Java, которые применяются для создания образов контейнеров и их развертывания в Kubernetes и OpenShift. Помимо программирования, любит делиться опытом работы на конференциях и в рассылках.

Об обложке

На обложке «Паттерны Kuberbetes» изображен красноносый нырок (*Netta rufina*). Название вида *rufina* на латыни означает «рыжий». Другое распространенное название — красноголовый нырок. «Нырок» означает «ныряющая утка». Красноносый нырок обитает в заболоченных местностях Европы и Центральной Азии. Его ареал обитания охватывает также болота Северной Африки и Южной Азии.

Длина тела взрослого красноносого нырка достигает 53–57 см, а вес — до 800–1250 г. Размах крыльев достигает почти одного метра. Самки имеют однородный бурый окрас перьев со светлой лицевой областью и менее красочны, чем самцы. Самцы красного нырка отличаются бурым окрасом головы, красным клювом, черной шеей, грудью и центральной частью брюха и белыми боками.

Рацион красноносого нырка состоит в основном из корней, семян и водных растений. Свои гнезда они устраивают в растительных зарослях рядом с болотами и озерами и откладывают яйца весной и летом. Обычно в выводке насчитывается 8–12 утят. Голос красноносые нырки подают в основном в период спаривания. Зов самца больше напоминает хрип, а зов самки звучит как отрывистое «вра, вра, вра».

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой вымирания; все они очень важны для биосферы. Чтобы узнать, чем вы можете помочь, посетите сайт animals.oreilly.com.

Иллюстрацию для обложки нарисовала Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры из «British Birds».