O'REILLY®

Kubernetes для DevOps

развертывание, запуск и масштабирование в облаке

Джон Арундел
Джастин Домингус

Cloud Native DevOps with Kubernetes

Building, Deploying, and Scaling Modern Applications in the Cloud

John Arundel and Justin Domingus

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Kubernetes для DevOps

развертывание, запуск и масштабирование в облаке

Джон Арундел
Джастин Домингус

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2020

ББК 32.988.02-018.2
УДК 004.451
А86

Арундел Джон, Домингус Джастин

А86 Kubernetes для DevOps: развертывание, запуск и масштабирование в облаке. — СПб.: Питер, 2020. — 384 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1602-7

Kubernetes — один из ключевых элементов современной облачной экосистемы. Эта технология обеспечивает надежность, масштабируемость и устойчивость контейнерной виртуализации. Джон Арундел и Джастин Домингус рассказывают об экосистеме Kubernetes и знакомят с проверенными решениями повседневных проблем. Шаг за шагом вы построите собственное облачно-ориентированное приложение и создадите инфраструктуру для его поддержки, настроите среду разработки и конвейер непрерывного развертывания, который пригодится вам при работе.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018.2
УДК 004.451

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1492040767 англ.

ISBN 978-5-4461-1602-7

Краткое содержание

Оглавление

Предисловие

Проект Kubernetes произвел настоящую революцию в индустрии. Чтобы понять важность Kubernetes в современных условиях, достаточно взглянуть на страницу Cloud Native Computing Foundation's Landscape (landscape.cncf.io), где представлены данные о более чем 600 проектах, существующих в облачном мире на сегодняшний день. Не все эти инструменты были разработаны для Kubernetes и не все из них можно использовать вместе с этой платформой, но все они являются частью огромной экосистемы, в которой Kubernetes — одна из ведущих технологий.

Появление платформы Kubernetes позволило изменить принцип разработки и администрирования приложений: в настоящее время это ключевой компонент в мире DevOps. Благодаря ей разработчики получают гибкость, а администраторы — свободу. Kubernetes можно использовать вместе с любым крупным облачным провайдером, в локальных средах с физической инфраструктурой, а также на отдельном компьютере разработчика. Стабильность, гибкость, мощные интерфейсы прикладного программирования (API), открытый исходный код и активное сообщество разработчиков — это не полный перечень достоинств, благодаря которым данный продукт стал отраслевым стандартом аналогично тому, как Linux стала стандартом в мире операционных систем.

Это замечательное руководство для тех, кто ежедневно работает с Kubernetes или только начинает с ней знакомиться. Джон и Джастин охватили все основные аспекты развертывания, конфигурирования и администрирования Kubernetes, а также учли передовой опыт разработки и выполнения приложений на этой платформе. В то же время вы получите отличный обзор сопутствующих технологий, включая Prometheus и Helm, а также изучите непрерывное развертывание. Таким образом, книга обязательна к прочтению для всех, кто работает в сфере DevOps.

Kubernetes не просто еще один интересный инструмент — это отраслевой стандарт и фундамент для технологий следующего поколения, таких как бессерверные платформы (OpenFaaS, Knative) и машинное обучение (Kubeflow). Благодаря облачной революции меняется вся индустрия информационных технологий, и быть частью этого процесса невероятно захватывающе.

Игорь Дворецкий,
Developer Advocate, Cloud Native Computing Foundation,
декабрь 2018

Введение

В мире компьютерного администрирования ключевые принципы DevOps являются общепонятными и общепринятыми. Но картина начинает меняться: по всему миру компании из самых различных отраслей внедряют новую платформу приложений под названием Kubernetes. И чем чаще приложения и организации переходят от традиционных серверов к среде Kubernetes, тем больше люди хотят знать, как реализовывать DevOps в этих новых условиях.

Прочитав книгу, вы узнаете, что такое DevOps в облачно-ориентированном мире, в котором Kubernetes служит стандартной платформой, а также сможете выбрать лучшие инструменты и фреймворки из экосистемы Kubernetes. Мы предложим понятные инструкции по использованию этих инструментов и фреймворков, базирующиеся на проверенных решениях, которые уже сейчас работают в реальных промышленных условиях.

Чему вы научитесь

Вы узнаете, что такое платформа Kubernetes, откуда она взялась и как в будущем она повлияет на администрирование и разработку ПО, как работают, собираются и управляются контейнеры, как проектировать облачно-ориентированные сервисы и инфраструктуру.

Вы оцените преимущества и недостатки самостоятельного построения и размещения кластеров Kubernetes и использования управляемых сервисов. Поймете возможности и ограничения, а также плюсы и минусы популярных инструментов для установки Kubernetes, таких как kops, kubeadm и Kubespray. Кроме того, получите обоснованный обзор основных решений для управления данной платформой от Amazon, Google и Microsoft.

Прочитав книгу, вы приобретете практический опыт написания и развертывания приложений Kubernetes, конфигурирования и администрирования кластеров, а так-

же автоматизации облачной инфраструктуры и развертывания с помощью такого инструмента, как Helm. Познакомитесь с системами безопасности, аутентификации и управления доступом, в том числе на основе ролей (RBAC), и с рекомендованными методиками защиты контейнеров и Kubernetes в промышленных условиях.

Вы научитесь настраивать непрерывные интеграцию и развертывание с помощью этой платформы, выполнять резервное копирование и восстановление данных, проверять свои кластеры на соответствие и надежность, мониторить, трассировать, заносить в журнал и агрегировать различные показатели. А также узнаете, как сделать инфраструктуру Kubernetes масштабируемой, устойчивой и экономичной.

Все вышесказанное в целях иллюстрации мы проверим на простом демонстрационном приложении. Следовать всем приводимым здесь примерам можно, используя код из нашего Git-репозитория.

Для кого предназначена книга

Книга наиболее актуальна для сотрудников отделов администрирования, ответственных за серверы, приложения и сервисы, а также для разработчиков, занимающихся либо построением новых облачных сервисов, либо миграцией существующих приложений в Kubernetes и облако. Не волнуйтесь, уметь работать с Kubernetes и контейнерами не требуется — мы всему научим.

Опытные пользователи Kubernetes также найдут для себя много ценного: здесь углубленно рассматриваются такие темы, как RBAC, непрерывное развертывание, управление конфиденциальными данными и наблюдаемость. Надеемся, что на страницах книги обязательно окажется что-нибудь интересное и для вас, независимо от ваших навыков и опыта.

На какие вопросы отвечает книга

Во время планирования и написания книги мы обсуждали облачные технологии и Kubernetes с сотнями людей, разговаривали как с лидерами и экспертами в данной отрасли, так и с абсолютными новичками. Ниже приведены отдельные вопросы, ответы на которые им хотелось бы увидеть в этом издании.

- «Меня интересует, почему следует тратить время на эту технологию. Какие проблемы она поможет решить мне и моей команде?»

- «Kubernetes кажется интересной, но имеет довольно высокий порог вхождения. Подготовить простой пример не составляет труда, но дальнейшие администрирование и отладка пугают. Мы бы хотели получить надежные советы о том, как люди управляют кластерами Kubernetes в реальных условиях и с какими проблемами мы, скорее всего, столкнемся».
- «Был бы полезен субъективный совет. Экосистема Kubernetes предлагает начинающим командам слишком много вариантов на выбор. Когда одно и то же можно сделать несколькими способами, как понять, какой из них лучше? Как сделать выбор?»

  И, наверное, наиболее важный из всех вопросов:

- «Как использовать Kubernetes, не нарушая работу моей компании?»

При написании книги мы помнили и об этих, и о других вопросах, стараясь как можно лучше на них ответить. Как мы справились с задачей? Чтобы узнать, читайте дальше.

Условные обозначения

В этой книге вы встретите следующие типографские обозначения.

Рубленый шрифт

Используется для выделения URL-адресов и адресов электронной почты.

Курсивный шрифт

Используется для выделения новых терминов и понятий, на которых сделан акцент.

`Моноширинный шрифт`

Используется для записи листингов программ, а также для выделения в тексте таких элементов, как имена переменных и функций, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена и расширения файлов.

`Полужирный моноширинный шрифт`

Используется для выделения команд или другого текста, который должен вводиться пользователем без каких-либо изменений.

Курсивный моноширинный шрифт

Используется для обозначения в коде элементов, которые требуется заменить предоставленными пользователем значениями или значениями, зависящими от контекста.

Этот значок обозначает совет или предложение.

Этот значок обозначает примечание общего характера.

Этот значок обозначает предупреждение или предостережение.

Использование примеров кода

Дополнительный материал (примеры кода, упражнения и т. д.) доступен для загрузки по адресу github.com/cloudnativedevops/demo.

Предназначение этой книги — помочь в решении ваших задач. Если предлагается какой-то пример кода, его разрешается использовать в ваших программах и документации. Связываться с нами, если вы не воспроизводите его существенную часть, не нужно: например, в ситуации, когда вы включаете в свою программу несколько фрагментов кода, приведенного здесь. Однако продажа или распространение CD-ROM с примерами из книг издательства O'Reilly требует отдельного разрешения. Цитировать эту книгу с примерами кода при ответе на вопрос вы можете свободно, но если хотите включить существенную часть приведенного здесь кода в документацию своего продукта, вам следует связаться с нами.

Мы приветствуем ссылку на оригинал, но не требуем ее. Ссылка обычно состоит из названия, имени автора, издательства, ISBN. Например, «“Kubernetes для DevOps: развертывание, запуск и масштабирование в облаке”, Джон Арундел и Джастин Домингус. Питер, 2020. 9785446116027».

Если вам кажется, что ваша работа с примерами кода выходит за рамки добросовестного использования или условий, перечисленных выше, можете обратиться к нам по адресу permissions@oreilly.com.

Благодарности

Мы благодарны многим людям, которые читали черновики этой книги и поделились с нами своими бесценными мнениями и советами, а также помогли нам каким-то другим образом (список может быть неполным): это Эбби Бангсер, Адам Дж. Макпартлен, Адриенна Домингус, Алексис Ричардсон, Эрон Трауринг, Камилла Монтонен, Габриель Насименто, Ханна Клемме, Ганс Финдель, Йан Кросби, Йан Шау, Игорь Дворецкий, Айк Деволдер, Джереми Йейтс, Джером Петаззони, Джессика Дин, Джон Харрис, Джон Барбер, Китти Карат, Марко Ланцини, Марк Элленс, Мэтт Норт, Мишель Бланк, Митчелл Кент, Николас Штейнметц, Найджел Браун, Патрик Дудич, Пол ван дер Линден, Филип Энсаргет, Пьетро Мамберти, Ричард Харпер, Рик Хайнесс, Сатьяджит Бхат, Суреш Вишной, Томас Лиакос, Тим Макгиннис, Тоби Сулливан, Том Холл, Винсент Де Смет и Уилл Теймс.

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 Революция в облаке

> У мира никогда не было момента начала, поскольку он идет по кругу, а у круга нет такого места, где бы он начинался.
>
> *Алан Уотс*

Прямо сейчас происходит революция. Точнее, три революции.

Первая — создание облака: мы объясним, что это такое и почему это важно.

Вторая — расцвет DevOps: вы узнаете, с чем это связано и как это меняет сферу системного администрирования.

Третья — появление контейнеров. Вместе они формируют новый мир программного обеспечения — *облачно-ориентированный* мир. И его операционная система называется Kubernetes.

В первой главе мы кратко изложим историю этих трех революций, объясним их значение и исследуем то, как они повлияли на способы развертывания и администрирования ПО. Мы расскажем, что понимается под *облачной ориентированностью* и каких изменений можно ожидать в этом новом мире, если ваша работа связана с разработкой, системным администрированием, развертыванием, проектированием, организацией сетей и безопасностью приложений.

Мы считаем, что благодаря эффектам от этих трех взаимосвязанных революций будущее компьютерных вычислений принадлежит облачным контейнеризированным распределенным системам, которые динамически управляются путем автоматизации на платформе Kubernetes (или на чем-то очень похожем). Искусство

разработки и выполнения этих приложений — *облачно-ориентированный DevOps* — это то, что мы будем исследовать на остальных страницах этой книги.

Если вы уже знакомы с историей и вам не терпится поработать с Kubernetes, можете спокойно переходить к главе 2. В противном случае усаживайтесь поудобнее с чашкой своего любимого напитка. Мы начинаем.

Создание облака

В самом начале (во всяком случае, в 1960-х) компьютеры заполняли собой бесконечные ряды стоек в огромных труднодоступных охлаждаемых вычислительных центрах. Пользователи не видели компьютеров и напрямую с ними не контактировали: отправляли задания и ждали результатов. Сотни и тысячи людей делили между собой одну и ту же вычислительную инфраструктуру и просто получали счета на суммы, соответствующие количеству потраченного на их задания процессорного времени или ресурсов.

Компаниям и организациям было невыгодно покупать и обслуживать собственное вычислительное оборудование, таким образом и сложилась бизнес-модель, в которой пользователи делили вычислительную мощь удаленных компьютеров, принадлежавших третьей стороне.

Может показаться, что мы говорим о современности, а не о прошлом столетии, и это не случайно. Слово «революция» означает «круговое движение», и индустрия вычислений в каком-то смысле вернулась к тому, с чего начиналась. С годами компьютеры стали куда более мощными (современные часы Apple Watch эквивалентны примерно трем мейнфреймам, изображенным на рис. 1.1), но разделяемый доступ к вычислительным ресурсам с платой за использование является очень старой идеей. В наши дни мы называем это *облаком*, и революция, у истоков которой стояли мейнфреймы с разделением времени, вернулась в исходную точку.

Покупаем время

Основная идея облака такова: вместо покупки *компьютера* вы покупаете *вычисления*. Иными словами, вместо крупных капиталовложений в физическое оборудование, которое сложно масштабировать, которое легко ломается и быстро устаревает, вы просто покупаете время на компьютере, принадлежащем другим людям, и снимаете с себя заботы о масштабировании, обслуживании и обновлении. Во времена «железных» серверов (назовем их «железным веком») вычислительная мощь требовала капитальных расходов. Теперь это операционные расходы, в чем и состоит разница.

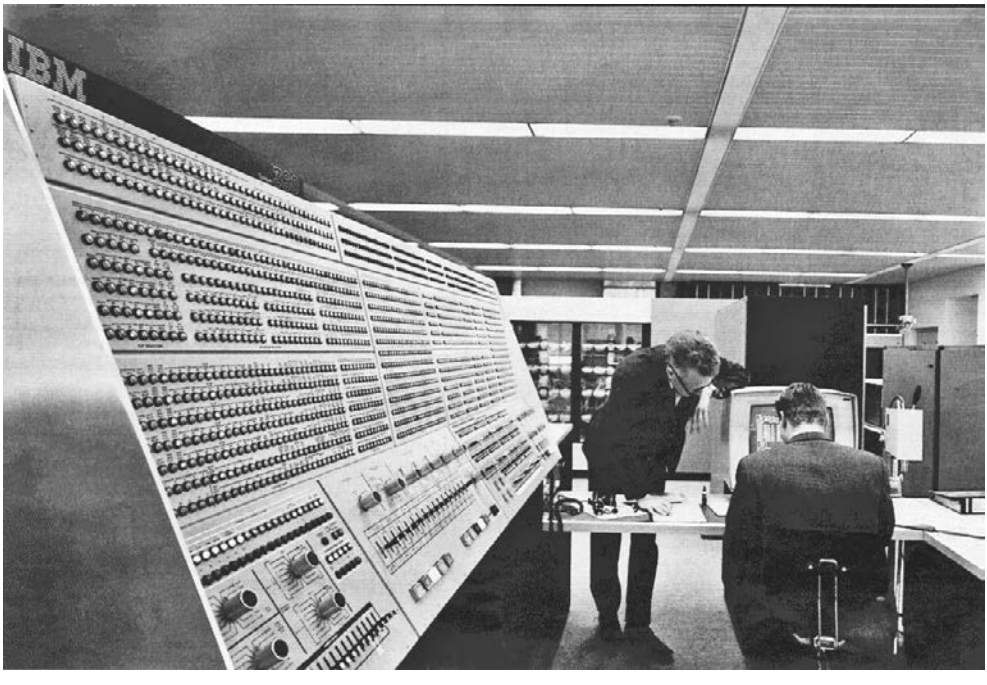

Рис. 1.1. Один из первых облачных компьютеров: IBM System/360 Model 91 в Центре космических полетов Годдарда, НАСА

Облако — это не просто удаленные арендуемые вычислительные ресурсы; это также распределенные системы. Вы можете купить непосредственно вычислительный ресурс (такой как сервер Google Compute или функция AWS Lambda) и использовать его для запуска собственного программного обеспечения, но все чаще вы будете арендовать *облачные сервисы*, а в сущности, доступ к чужому ПО. Например, если вы применяете PagerDuty для мониторинга своих систем и оповещения в случае каких-то сбоев, вы уже используете облачный сервис — то, что называется *программное обеспечение как услуга* (software as a service, SaaS).

Инфраструктура как услуга

Когда вы используете облачную инфраструктуру для запуска собственных сервисов, вы покупаете *инфраструктуру как услугу* (infrastructure as a service, IaaS). Она не требует капиталовложений, вам не нужно ее собирать или обновлять. Это всего лишь товар — как электричество или вода. Облачные вычисления совершили революцию во взаимоотношениях между бизнесом и его информационной инфраструктурой.

Аутсорсингом аппаратного обеспечения дело не ограничивается. Облако также позволяет выполнять аутсорсинг *программного обеспечения*, которое вы не пишете сами (операционные системы, базы данных), обеспечивать высокую доступность, осуществлять кластеризацию, репликацию, мониторинг, работать с сетью, обрабатывать очереди и потоки и все те мириады программных и конфигурационных слоев, которые заполняют собой промежуток между вашим кодом и ЦПУ. Управляемые сервисы могут взять на себя практически все эти *тяжкие обязанности* (больше о преимуществах управляемых сервисов вы узнаете из главы 3).

Облачная революция полностью изменила то, как люди используют облака: речь идет о движении DevOps.

Расцвет DevOps

До появления DevOps разработка и системное администрирование ПО представляли собой, в сущности, два отдельных направления, и занимались ими две разные группы людей. *Разработчики* (developers) писали код и передавали его коллегам-*администраторам* (operations), которые занимались запуском и эксплуатацией этого кода (то есть предоставляли услуги реальным пользователям, а не просто работали в режиме тестирования). Как и размещение компьютеров на отдельном этаже, это разделение корнями уходит в середину прошлого века. Написание программного обеспечения было узкоспециализированным направлением, равно как и администрирование компьютеров. И пересекались они редко.

Действительно, эти два направления имели довольно разные цели и мотивации, которые часто противоречили друг другу (рис. 1.2). Разработчикам свойственно фокусироваться на быстрой реализации новых возможностей, тогда как команды системных администраторов заботятся о том, чтобы сделать сервисы стабильными и надежными в долгосрочной перспективе.

Ситуация изменилась, когда на горизонте появилось облако. Распределенные системы имеют высокую сложность, а Интернет безграничный. Нюансы управления системой (восстановление после сбоев, реагирование по истечении времени ожидания, плавный переход на новые версии) не так-то легко отделить от ее проектирования, архитектуры и реализации.

Более того, в наши дни под системой подразумевается не только ваш код: это и внутреннее ПО, и облачные сервисы, и сетевые ресурсы, и балансировщики нагрузки, и инструменты мониторинга, и сети доставки содержимого, и брандмауэры, и DNS и т. д. Все эти компоненты тесно связаны между собой и зависят друг от друга.

Рис. 1.2. У разных команд могут быть противоположные мотивы (фото Дейва Рота)

Люди, пишущие программное обеспечение, должны понимать, как оно соотносится с остальной частью системы, а люди, администрирующие систему, должны понимать, как это ПО работает (или не работает).

Движение DevOps началось с попыток свести две группы воедино: для сотрудничества, налаживания взаимопонимания, разделения ответственности за надежность системы и корректность программного кода, для улучшения масштабируемости как программного обеспечения, так и команд разработчиков, которые его пишут.

Никто не понимает DevOps

Иногда DevOps считают спорной идеей. Кто-то настаивает на том, что это не более чем современное название для уже существующих проверенных методик в разработке ПО, а кто-то отрицает необходимость более тесного взаимодействия между разработчиками и системными администраторами.

Кроме того, по-прежнему нет окончательного понимания, что же такое на самом деле DevOps: должность? Методология? Набор навыков? Джон Уиллис, авторитетный автор книг о DevOps, определяет четыре ключевых аспекта, на которых

основана данная концепция: культура, автоматизация, измерение и обмен знаниями (culture, automation, measurement, sharing, вместе CAMS). Брайан Доусон предлагает другое определение, которое он называет троицей DevOps: люди и культура, процесс и практика, инструменты и технологии.

Кто-то считает, что благодаря облаку и контейнерам DevOps нам больше не нужен — эту точку зрения иногда называют *NoOps*. Ее суть в том, что все системное администрирование делегируется облачному провайдеру или другому стороннему сервису, поэтому компаниям не нужны штатные системные администраторы.

Ошибочность этого мнения заключается в неправильном понимании того, что на самом деле входит в работу DevOps.

> Благодаря DevOps большая часть традиционного системного администрирования выполняется до того, как код достигает промышленной среды. Выпуск каждой версии подразумевает мониторинг, ведение журнала и A/B-тестирование. Конвейеры CI/CD автоматически выполняют модульные тесты, сканирование безопасности и проверку политик при каждой фиксации изменений. Развертывание происходит автоматически. Контроль, задачи и нефункциональные требования теперь прорабатываются перед выпуском версии, а не во время или после критического сбоя в работе.
>
> *Джордан Бах (AppDynamics)*[1]

Важнейшая деталь относительно DevOps, которую необходимо усвоить, состоит в том, что такая практика зависит прежде всего от человеческого, а не от технического фактора. Что согласуется со *вторым законом консалтинга* Джерри Вайнберга.

> Неважно, как это выглядит на первый взгляд, но проблема всегда в людях.
>
> *Джеральд М. Вайнберг. Закон малинового варенья и еще 103 секрета консалтинга*

Бизнес-преимущество

С точки зрения бизнеса DevOps имеет следующее описание: «Улучшение качества ПО путем ускорения цикла выпуска версий с помощью облачных методик и автоматизации и с дополнительным преимуществом в виде того, что программное обеспечение на самом деле продолжает работать в промышленных условиях» (The Register, www.theregister.co.uk/2018/03/06/what_does_devops_do_to_decades_old_planning_processes_and_assumptions).

---

[1] www.appdynamics.com/blog/engineering/is-noops-the-end-of-devops-think-again.

Внедрение DevOps требует от компании глубокого культурного преобразования, которое должно начинаться на руководящем, стратегическом уровне и постепенно распространяться на каждый отдел организации. Скорость, динамичность, взаимодействие, автоматизация и качество ПО являются ключевыми целями DevOps, что для многих компаний означает существенные изменения в образе мышления.

Но DevOps работает. Проводимые исследования указывают на то, что организации, внедрившие принципы DevOps, быстрее выпускают качественный код, легче и своевременнее реагируют на сбои и проблемы, показывают лучшую динамику на рынке и существенно повышают качество своих продуктов.

> DevOps — это не причуда, а, скорее, способ, с помощью которого успешные организации в наши дни индустриализируют доставку качественного программного обеспечения; уже завтра и на ближайшие годы это станет новой нормой.
>
> *Брайан Доусон (Cloudbees) (Computer Business Review)*[1]

Инфраструктура как код

Когда-то давным-давно разработчики занимались программным обеспечением, а команды системных администраторов отвечали за оборудование и операционные системы, которые на нем работают.

Теперь, когда оборудование находится в облаке, все в каком-то смысле перешло на программный уровень. Движение DevOps привносит навыки написания ПО в мир системного администрирования: дает инструменты и рабочие процессы для быстрого и динамичного совместного построения сложных систем. С этой концепцией неразрывно переплетается идея *инфраструктуры как кода*.

Вместо того чтобы физически размещать компьютеры и коммутаторы на стойках и прокладывать к ним кабели, облачную инфраструктуру можно выделять автоматически программным способом. Вместо того чтобы вручную устанавливать и обновлять аппаратное обеспечение, системные администраторы теперь пишут код, который автоматизирует облако.

Но это не односторонний процесс. Разработчики учатся у команд администраторов, как предупреждать сбои и проблемы, неизбежные в распределенных облачных системах, как смягчать их последствия и как проектировать программы, способные самостоятельно понижать свою производительность и контролируемо отключаться.

---

[1] www.cbronline.com/enterprise-it/applications/devops-fad-stay.

Учимся вместе

Команды разработчиков и системных администраторов учатся работать вместе: это касается проектирования и построения систем, мониторинга, а также получения информации о системах, работающих в промышленной среде, для их дальнейшего совершенствования. Что еще более важно — они учатся укреплять взаимодействие своих пользователей и давать больше отдачи бизнесу, который их финансирует.

Огромный масштаб облачных вычислений и совместная, ориентированная на код природа движения DevOps перевели системное администрирование в область программного обеспечения. Вместе с тем ПО теперь является заботой администраторов. Все это вызывает следующие вопросы.

- Как развертывать и обновлять программное обеспечение в огромных разнородных сетях с разными серверными архитектурами и операционными системами?
- Как выполнять развертывание в распределенных средах надежным и воспроизводимым образом, используя в основном стандартизированные компоненты?

Мы подошли к третьей революции: к появлению контейнеров.

Пришествие контейнеров

Чтобы развернуть программный компонент, вам нужен не только он сам, но и его *зависимости*, то есть библиотеки, интерпретаторы, дочерние пакеты, компиляторы, расширения и т. д.

Нужна и его *конфигурация*: настройки, подробности, относящиеся к определенному сайту, лицензионные ключи, пароли к базам данных — все, что превращает сырое программное обеспечение в полезный сервис.

Последние веяния

Более ранние попытки решить эту проблему включали в себя использование систем *управления конфигурациями*, таких как Puppet и Ansible, которые состояли из кода для установки, запуска, настройки и обновления поставляемого ПО.

В качестве альтернативы некоторые языки предоставляют собственные механизмы управления пакетами, такие как JAR-файлы в Java, eggs в Python и gems в Ruby.

Однако они работают в пределах одного конкретного языка и не могут полностью решить проблему зависимостей — например, прежде, чем вы сможете запустить JAR-файл, вам все равно нужно установить среду выполнения Java.

Еще одним решением является *универсальный (omnibus) пакет*, который, как следует из его названия, пытается вместить все нужное приложению в единый файл. Такой пакет содержит программное обеспечение, его конфигурацию, программные компоненты, от которых оно зависит, *их* конфигурацию, *их* зависимости и т. д. (например, универсальный пакет Java содержал бы среду выполнения JRE, а также все JAR-файлы для приложения).

Некоторые поставщики пошли еще дальше — начали предоставлять целую компьютерную систему, необходимую для запуска приложения, в виде *образа виртуальной машины*. Но такая система слишком громоздкая, занимает много места и времени для построения и обслуживания, легко ломается во время работы, медленно загружается и развертывается, а также является крайне неэффективной с точки зрения производительности и потребления ресурсов.

Как системному администратору, вам придется управлять не только разными форматами пакетов, но и целой флотилией серверов, на которых они будут выполняться.

Серверы нужно выделять, подключать к сети, развертывать, конфигурировать, поддерживать в актуальном состоянии с помощью патчей безопасности, мониторить, администрировать и т. д.

Все это требует значительных затрат времени, навыков и усилий — и только для того, чтобы предоставить платформу для выполнения программного обеспечения. Неужели нет лучшего способа?

«Коробочное» мышление

Чтобы решить эти проблемы, компьютерная индустрия позаимствовала у индустрии судоходства концепцию *контейнера*. В 1950-х годах дальнобойщик по имени Малколм Маклин предложил следующую идею: вместо того чтобы последовательно выгружать отдельные товары из грузовых прицепов, в которых они доставлялись в морской порт, а затем переправлять их на корабли, лучше просто грузить на корабли сами грузовики — или, точнее, их прицепы.

Грузовой прицеп, по сути, является большим металлическим ящиком на колесах. Если отделить этот ящик (контейнер) от колес и шасси, на котором он транспортируется, его можно будет легко поднимать, загружать, складывать

и разгружать. Кроме того, его можно поместить прямо на корабль или другой грузовик (рис. 1.3).

Фирма Маклина по транспортировке контейнеров Sea-Land внедрила эту систему и стала очень успешной благодаря значительному снижению стоимости на доставку товаров. Контейнеры быстро приобрели популярность (www.freightos.com/the-history-of-the-shipping-container). Сегодня каждый год перевозятся сотни миллионов контейнеров с товарами стоимостью триллионы долларов.

Рис. 1.3. Контейнеры существенно снизили стоимость транспортировки товаров в больших объемах (фото Pixabay, www.pexels.com/@pixabay, под лицензией Creative Commons 2.0)

Размещение программного обеспечения в контейнерах

Программные контейнеры основаны на точно такой же идее — это стандартный универсальный и широко распространенный формат упаковки и доставки, который позволяет существенно увеличить пропускную способность, снизить денежные затраты, сэкономить на масштабе и упростить обслуживание. Формат контейнеров

предусматривает все, что необходимо приложению для его работы, помещая это в единый *файл образа*, который может быть запущен *средой выполнения контейнеров*.

Чем это отличается от образа виртуальной машины? Тот тоже содержит все необходимое для работы приложения — но еще и много чего сверх. Типичный образ занимает около 1 ГиБ[1]. Для сравнения: хорошо спроектированный образ контейнера может оказаться в сто раз меньше.

Поскольку виртуальная машина (ВМ) содержит множество посторонних программ, библиотек и компонентов, которые ваше приложение никогда не будет использовать, большая часть ее места расходуется впустую. Передача образов ВМ по сети намного медленнее по сравнению с оптимизированными контейнерами.

Что еще хуже, виртуальные машины по-настоящему *виртуальные*: внутреннее физическое оборудование, в сущности, реализует *эмулируемый* центральный процессор, на котором выполняется виртуальная машина. Слой виртуализации оказывает существенное отрицательное воздействие на производительность (www.stratoscale.com/blog/containers/running-containers-on-bare-metal): при тестировании виртуализированные рабочие задания выполняются на 30 % медленнее, чем в аналогичных контейнерах.

Для сравнения: контейнеры работают непосредственно на реальном ЦПУ и без дополнительных расходов на виртуализацию — точно так же, как обычные исполняемые файлы.

И поскольку контейнеры содержат только нужные им файлы, их размер куда меньше, чем у образов ВМ. Они также используют продуманную методику адресуемых *слоев* файловой системы, которые можно разделять и задействовать в разных контейнерах.

Например, если у вас есть два контейнера, каждый из которых основан на базовом образе Debian Linux, этот базовый образ нужно будет загрузить лишь один раз, и тогда оба контейнера смогут на него ссылаться.

Среда выполнения контейнеров соберет все необходимые слои и загрузит только те из них, которые еще не закэшированы локально. Это делает использование дискового пространства и пропускной способности сети крайне эффективным.

---

[1] Гибибайт (ГиБ) — единица измерения данных, одобренная Международной электротехнической комиссией (International Electrotechnical Commission, IEC). Равна 1024 мебибайтам (МиБ). Чтобы избежать недопонимания, в этой книге мы используем единицы измерения IEC (ГиБ, МиБ, КиБ).

Динамически подключаемые приложения

Контейнер — это единица не только развертывания и упаковки, но еще и *повторного использования* (один и тот же образ контейнера можно использовать в качестве компонента в разных сервисах), *масштабирования* и *выделения ресурсов* (контейнер может работать где угодно при наличии достаточного количества ресурсов, удовлетворяющих его конкретным потребностям).

Разработчикам больше не нужно волноваться о поддержке разных версий ПО, запущенных на дистрибутивах Linux с разными версиями библиотек и языков, и т. д. Единственное, от чего зависят контейнеры, — это ядро операционной системы (например, Linux).

Просто поместите свое приложение в образ контейнера, и оно сможет работать на любой платформе, которая поддерживает стандартный формат контейнеров и имеет совместимое ядро.

В своей статье «Шаблоны проектирования для контейнерных распределенных систем» (https://www.usenix.org/node/196347) разработчики Kubernetes Брендан Бернс и Дэвид Оппенгеймер выразили эту концепцию следующим образом:

> Будучи герметично запечатаны, сохраняя в себе свои зависимости и предоставляя атомарный сигнал развертывания («успех»/«провал»), контейнеры значительно улучшили средства для развертывания ПО в вычислительных центрах и облаке. Однако контейнеры имеют потенциал стать чем-то значительно большим, чем просто усовершенствованный механизм развертывания, — мы считаем, что им суждено быть аналогом объектов в объектно-ориентированных программных системах. Тем самым они приведут к развитию шаблонов проектирования распределенных систем.

Дирижирование оркестром контейнеров

Команды системных администраторов также замечают существенное упрощение своих рабочих процессов благодаря контейнерам. Вместо того чтобы обслуживать обширный парк разнообразных компьютеров с разными архитектурами и операционными системами, теперь достаточно запустить *оркестратор контейнеров*: программный компонент, предназначенный для объединения множества разных серверов в *кластер*. Кластер — это своего рода унифицированная вычислительная подложка, которая, с точки зрения пользователя, выглядит как компьютер очень высокой мощности, способный работать с контейнерами.

Термины *«оркестрация»* и *«планирование»* часто используются как синонимы. Хотя, строго говоря, в этом контексте *оркестрация* означает координацию и выстраивание цепочки из разных действий для достижения общей цели (как это делают музыканты в оркестре). *Планирование* же означает управление доступными ресурсами и направление рабочих заданий туда, где они могут быть выполнены наиболее эффективно (здесь вовсе не имеется в виду, что каждое задание должно быть выполнено в указанное время).

Третьей важной задачей *управления кластером* является объединение нескольких физических или виртуальных серверов в унифицированную, надежную, отказоустойчивую и довольно цельную группу.

Термин *«оркестратор контейнеров»* обычно относится к одному сервису, который занимается планированием и оркестрацией кластера, а также управлением им.

Контейнеризация (использование контейнеров в качестве стандартного метода развертывания и запуска ПО) имеет очевидные преимущества, а стандарт дефакто формата контейнеров позволяет экономить на масштабе самыми разными способами. При всем этом на пути широкого внедрения контейнеров стояла одна преграда: отсутствие стандартной системы оркестрации.

Пока на рынке конкурировали несколько разных инструментов для планирования и оркестрации контейнеров, компании не горели желанием рисковать крупными суммами и делать ставку на одну из них. Но вскоре ситуация изменилась.

Kubernetes

Компания Google использовала контейнеры в крупных масштабах и с промышленными нагрузками задолго до кого бы то ни было. Почти все ее сервисы работали в контейнерах: Gmail, Google Search, Google Maps, Google App Engine и т. д. Поскольку в то время не существовало подходящей системы оркестрации контейнеров, разработчикам Google пришлось ее изобрести.

От Borg до Kubernetes

Чтобы решить проблему запуска большого количества сервисов в глобальном масштабе на миллионах серверов, компания Google разработала закрытую, внутреннюю систему оркестрации контейнеров под названием Borg (pdos.csail.mit.edu/6.824/papers/borg.pdf).

По своей сути Borg — это централизованная и очень мощная система управления, которая определяет место и планирует выполнение контейнеров в пуле серверов. Она тесно связана с внутренними и фирменными компонентами Google, поэтому ее сложно расширять и невозможно сделать публичной.

В 2014 году компания Google основала открытый проект под названием Kubernetes (от греческого слова κυβερνήτης — «рулевой, пилот»). В его рамках планировалось разработать оркестратор контейнеров, доступный для всеобщего использования, который был бы основан на уроках, извлеченных из деятельности Borg и ее преемницы Omega (ai.google/research/pubs/pub41684.pdf).

Восход Kubernetes был ошеломляющим. Системы оркестрации контейнеров существовали и раньше, но это были коммерческие продукты, привязанные к поставщикам, что всегда служило преградой на пути к их всеобщему внедрению. С приходом по-настоящему свободного и открытого оркестратора внедрение как контейнеров, так и Kubernetes росло феноменальными темпами.

К концу 2017 года война оркестраторов закончилась победой Kubernetes. И хотя другие системы все еще используются, компаниям, желающим перевести свою инфраструктуру на контейнеры, теперь стоит рассматривать лишь одну платформу — Kubernetes.

Что делает платформу Kubernetes такой ценной

Келси Хайтауэр, штатный сотрудник Google, соавтор книги *Kubernetes Up & Running* (O'Reilly) и просто легенда в сообществе Kubernetes, дает такой ответ:

> Kubernetes занимается тем же, чем и самые лучшие системные администраторы: автоматизацией, централизованным ведением журнала, мониторингом, обеспечивает отказоустойчивость. Данная платформа берет то, чему мы научились в сообществе DevOps, и делает из этого готовое решение по умолчанию.

Многие традиционные задачи системного администрирования, такие как обновление серверов, установка патчей безопасности, настройка сетей и выполнение резервного копирования, являются не самыми приоритетными в облачно-ориентированном мире. Kubernetes способна автоматизировать все эти действия, чтобы ваша команда могла сосредоточиться на выполнении основной работы.

Некоторые из этих возможностей, такие как балансировка нагрузки и автомасштабирование, встроены в ядро Kubernetes, другие предоставляются в виде дополнений, расширений и сторонних инструментов с использованием стандартного API. Kubernetes отличается обширной и постоянно растущей экосистемой.

Kubernetes облегчает развертывание

За это платформу любят администраторы. Однако и разработчикам она дает некоторые существенные преимущества: значительно сокращает затраты времени и сил на развертывание. Благодаря тому что Kubernetes выполняет плавающие обновления по умолчанию, приобрели популярность развертывания с нулевым временем простоя (сначала запускаются контейнеры с новой версией, и после того, как они станут работоспособными, выключаются старые версии).

Kubernetes также предоставляет средства, которые помогают реализовать методики непрерывного развертывания, такие как *канареечные обновления*: речь о постепенном выкатывании обновлений сервер за сервером для выявления проблем на ранних стадиях (см. подраздел «Канареечные развертывания» на с. 306). Еще одной общепринятой практикой являются сине-зеленые развертывания: параллельный запуск новой версии системы и переключение трафика по мере ее готовности (см. подраздел «Сине-зеленые развертывания» на с. 305).

Скачки нагрузки больше не выведут ваш сервис из строя, потому что Kubernetes поддерживает автомасштабирование. Например, если использование ЦПУ контейнером достигнет определенного уровня, система сможет добавлять новые копии этого контейнера до тех пор, пока показатель не опустится ниже предельного значения. Когда нагрузка снизится, Kubernetes снова уменьшит количество копий, освобождая ресурсы кластера для выполнения других задач.

Поскольку избыточность и отказоустойчивость встроены в Kubernetes, ваше приложение будет более надежным и устойчивым. Некоторые управляемые сервисы могут даже масштабировать вверх и вниз сам кластер Kubernetes в зависимости от нагрузки: благодаря этому вы всегда будете платить только за те ресурсы, которые вам нужны в тот или иной момент (см. пункт «Автомасштабирование» на с. 145).

Бизнесу эта платформа тоже придется по душе, так как она снижает затраты на инфраструктуру и куда более эффективно потребляет имеющиеся ресурсы. Традиционные и даже облачные серверы простаивают большую часть времени. Дополнительная емкость, которая нужна для того, чтобы справиться со всплесками нагрузки, в обычных условиях по большому счету тратится впустую.

Kubernetes берет эту емкость и использует ее для выполнения рабочих заданий, позволяя вам добиться гораздо большей загруженности серверов. При этом вы еще получаете масштабирование, балансировку нагрузки и отказоустойчивость.

Некоторые из этих возможностей, такие как автомасштабирование, были доступны и до Kubernetes, однако они всегда были привязаны к отдельному облачному

провайдеру или сервису. Kubernetes *не зависит от провайдеров*: определившись с нужными вам ресурсами, вы можете выполнять их на любом кластере Kubernetes, на каком бы облаке он ни работал.

Это не означает, что Kubernetes предоставляет лишь необходимый минимум, — он связывает ваши ресурсы с соответствующими возможностями того или иного провайдера. Например, если это Google Cloud, то сервис Kubernetes с балансировкой нагрузки создаст балансировщик GCE, а если это Amazon — балансировщик AWS. Kubernetes абстрагирует детали, присущие конкретному провайдеру, и позволяет вам сосредоточиться на описании поведения вашего приложения.

Не исчезнет ли Kubernetes?

Как ни странно, несмотря на нынешний энтузиазм вокруг Kubernetes, в ближайшие годы разговоры об этой платформе могут поутихнуть. Многие вещи, которые когда-то были новыми и революционными, сейчас стали просто частью компьютерных систем, и мы о них больше не задумываемся. Можно вспомнить микропроцессоры, манипулятор мыши, Интернет...

Kubernetes тоже, вероятно, потеряет актуальность, превратившись в тривиальный и скучный элемент инфраструктуры — в хорошем смысле этих слов: вам достаточно будет лишь научиться развертывать приложение в Kubernetes.

Будущее этой платформы, скорее всего, лежит в области управляемых сервисов. Виртуализация, которая когда-то была новой и захватывающей технологией, превратилась в обычный полезный инструмент. Большинство людей арендуют виртуальные машины у облачного провайдера, вместо того чтобы использовать собственную систему виртуализации, такую как vSphere или Hyper-V.

Точно так же Kubernetes станет настолько стандартной частью инфраструктуры, что о ее наличии внутри вы даже не будете знать.

Kubernetes не решает все проблемы

Будет ли инфраструктура будущего полностью базироваться на Kubernetes? Наверное, нет. Прежде всего, Kubernetes просто не очень хорошо подходит для некоторых структур (таких как базы данных).

> Оркестрация программного обеспечения в контейнерах подразумевает запуск новых взаимозаменяемых серверов без необходимости в координации между ними. Однако

> реплики баз данных не являются взаимозаменяемыми — каждая из них обладает уникальным состоянием, и развертывание такой реплики требует координации с другими узлами, чтобы такие действия, как изменение схемы, везде происходили одновременно.
>
> *Шон Луизель (Cockroach Labs)*[1]

В Kubernetes вполне возможно выполнять рабочие задания с сохранением состояния, получая при этом надежность на корпоративном уровне. Но это требует времени и инженерных усилий, которые могут оказаться чрезмерными для вашей компании (см. подраздел «Не используйте слишком много ПО» на с. 82). Обычно лучшим решением является использование управляемых сервисов.

К тому же некоторые задачи не требуют Kubernetes и могут работать на платформах, которые иногда называют *бессерверными* (serverless). У них есть более подходящее название: *функции как сервис* (functions as a service, FaaS).

Облачные функции и фунтейнеры

В качестве примера платформы FaaS можно привести AWS Lambda. Она позволяет выполнять код, написанный на Go, Python, Java, Node.js, C# и других языках, не требуя при этом никакой компиляции или развертывания вашего приложения. Amazon сделает это все за вас.

Поскольку вы платите за время выполнения по интервалам длиной 100 миллисекунд, модель FaaS идеально подходит для вычислений, которые производятся только тогда, когда они требуются. Вам не нужно платить за постоянно запущенный облачный сервер, если вы его не используете.

Облачные функции в некоторых случаях более удобны, чем контейнеры (хотя контейнеры могут работать и на некоторых платформах FaaS). Но лучше всего они подходят для коротких автономных задач (например, AWS Lambda ограничивает время выполнения функций 15 минутами, позволяя разворачивать файлы общим размером примерно 50 Мбайт), особенно если те интегрированы с существующими облачными вычислительными сервисами, такими как Microsoft Cognitive Services или Google Cloud Vision API.

Почему нам не нравится называть эту модель *бессерверной*? Потому что она таковой не является: вы по-прежнему используете сервер, просто не свой, а чужой.

---

[1] https://www.cockroachlabs.com/blog/kubernetes-state-of-stateful-apps.

Суть в том, что этот сервер вам не нужно выделять и обслуживать — облачный провайдер берет работу на себя.

Конечно же, не всякое рабочее задание подходит для выполнения на платформах FaaS, однако в будущем данная технология, вероятно, станет ключевой для облачно-ориентированных приложений.

Следует также отметить, что облачные функции не ограничиваются публичными платформами FaaS, такими как Lambda или Azure Functions: если у вас уже есть кластер Kubernetes и вы хотите запускать на нем FaaS-приложения, это можно сделать с помощью OpenFaaS (www.openfaas.com) и других открытых проектов. Такой гибрид функций и контейнеров иногда называют *фунтейнерами* (funtainers). Нам этот термин кажется симпатичным.

У Kubernetes есть более продвинутая платформа доставки ПО — Knative, которая вобрала в себя как контейнеры, так и облачные функции (см. подраздел «Knative» на с. 302). На сегодняшний день она находится на стадии активной разработки. Это многообещающий проект, который в будущем может размыть или вовсе стереть границу между контейнерами и функциями.

Облачная ориентированность

Термин *«облачно-ориентированный»* (cloud native) становится все более популярным сокращением, которое описывает современные приложения и сервисы, пользующиеся преимуществами облаков, контейнеров и оркестрации (часто с открытым исходным кодом).

В самом деле, в 2015 году была основана организация Cloud Native Computing Foundation, или CNCF (www.cncf.io), обязанная, по утверждению основателей, «сформировать сообщество вокруг набора высококачественных проектов, которые оркестрируют контейнеры в рамках микросервисной архитектуры».

Являясь частью Linux Foundation, CNCF объединяет разработчиков, конечных пользователей и поставщиков, в том числе основных публичных облачных провайдеров. Самым известным проектом под эгидой CNCF является сама Kubernetes, но эта организация также взращивает и продвигает множество других ключевых компонентов облачно-ориентированной экосистемы: Prometheus, Envoy, Helm, Fluentd, gRPC и т. д.

Так что же на самом деле имеется в виду под *облачной ориентированностью*? Как часто бывает, один термин может иметь различные значения для разных людей. Но, возможно, нам удастся найти нечто общее.

Облачно-ориентированные приложения работают в облаке, с этим никто не спорит. Но если мы возьмем существующее приложение и просто запустим его на облачном вычислительном сервере, от этого оно не станет облачно-ориентированным. То же самое можно сказать о запуске в контейнере или использовании таких облачных сервисов, как Cosmos DB от Azure или Pub/Sub от Google. Хотя это можно назвать важными аспектами облачной ориентированности.

Рассмотрим несколько характеристик облачно-ориентированных систем, с которыми может согласиться большинство людей.

- *Автоматизируемость*. Если приложения должны развертываться и управляться компьютерами, а не людьми, им следует использовать общепринятые стандарты, форматы и интерфейсы. Kubernetes предоставляет эти стандартные интерфейсы таким образом, что разработчики приложений могут о них не беспокоиться.
- *Универсальность и гибкость*. Поскольку контейнеризированные микросервисы отвязаны от физических ресурсов, таких как диски, и не имеют никакой конкретной информации о вычислительных узлах, на которых они работают, их можно легко переносить с узла на узел или даже между кластерами.
- *Устойчивость и масштабируемость*. У традиционных приложений обычно есть единая точка отказа: программа перестает работать в случаях сбоя в главном процессе, неполадок в оборудовании компьютера или перегруженности сети. Облачно-ориентированные приложения являются распределенными по своей природе, поэтому их можно сделать высокодоступными с помощью избыточности и плавной деградации.
- *Динамичность*. Система оркестрации, такая как Kubernetes, может планировать контейнеры для максимально эффективного использования доступных ресурсов. Она может запускать множество копий контейнера, чтобы достичь высокой доступности, и выполнять плавающие обновления, чтобы плавно переходить на новые версии сервисов, не теряя трафика.
- *Наблюдаемость*. Облачно-ориентированные приложения по своей природе труднее поддаются исследованию и отладке. Поэтому *наблюдаемость* является ключевым требованием к распределенным системам: мониторинг, ведение журнала, трассирование и сбор показателей помогают инженерам понять, чем занимаются их системы (и что они делают не так).
- *Распределенность*. Облачная ориентированность — это подход к построению и выполнению приложений, основанный на использовании преимущества от распределенной и децентрализованной природы облаков. Он определяет то, как ваше приложение работает, а не то, где оно запущено. Вместо развертывания кода в виде единой сущности, известной как *монолит*, облачно-ориентированные

приложения обычно состоят из нескольких распределенных *микросервисов*, которые взаимодействуют между собой. Микросервис — это просто самодостаточный сервис, делающий что-то одно. Если соединить необходимое количество микросервисов, получится приложение.

Не все сводится к микросервисам. Несмотря на все вышесказанное, использование микросервисов не является панацеей. В монолите проще разобраться, поскольку все находится в одном месте и можно отследить взаимодействие разных компонентов. Но монолит сложно масштабировать как с точки зрения самого кода, так и в отношении команд разработчиков, которые его обслуживают. По мере того как код разрастается, взаимодействие между разными его частями становится все более интенсивным и система в целом достигает масштабов, при которых отдельный человек не способен постичь ее целиком.

Хорошо спроектированное облачное приложение состоит из микросервисов. Но не так-то просто решить, что эти микросервисы должны собой представлять, где должны проходить их границы и как они должны между собой взаимодействовать. Хорошая архитектура облачно-ориентированного приложения требует принятия взвешенных решений о том, как разделить разные компоненты. Но даже при удачном проектировании облачно-ориентированная система остается распределенной, что изначально делает ее сложной, проблемной с точки зрения наблюдения и понимания, а также подверженной сбоям в неожиданных ситуациях.

Облачно-ориентированные системы обычно являются распределенными, но с помощью контейнеров в облаке можно выполнять и монолит, что само по себе оказывается существенным плюсом для бизнеса. Это может стать или первым шагом на пути к постепенной миграции элементов монолита вовне и превращению их в современные микросервисы, или временной мерой на период перепроектирования системы, пока она полностью не превратится в облачно-ориентированную.

Будущее системного администрирования

Системное администрирование и проектирование инфраструктуры — это работа, требующая высокой квалификации. Ставит ли ее под угрозу облачно-ориентированное будущее? Мы считаем, что нет.

Вместо этого квалификация станет еще более важной. Проектирование и понимание распределенных систем — сложная задача, сети и оркестраторы контейнеров тоже непростые темы. Любой команде, разрабатывающей облачно-ориентированные приложения, понадобятся навыки и знания системного администрирования. Автоматизация освобождает сотрудников от скучной, повторяемой ручной работы

и дает им время на решение более сложных, интересных и увлекательных задач, с которыми компьютеры еще не научились справляться самостоятельно.

Это не значит, что все текущие позиции системных администраторов останутся нетронутыми. Когда-то сисадминам не требовались навыки программирования, если не считать программированием создание крайне простых скриптов командной оболочки. В облаке такое не пройдет.

В мире, где программное обеспечение играет центральную роль, способность писать, понимать и поддерживать код становится критически важной. Тот, кто не может или не желает учиться чему-то новому, остается за бортом — и так было всегда.

Распределенный DevOps

Мы не создаем отдельную команду системных администраторов, обслуживающую другие команды, а распределяем ее обязанности между разными группами сотрудников.

У каждой команды разработчиков должен быть как минимум один специалист, отвечающий за работоспособность систем и сервисов, которые эта команда предоставляет. Он также будет заниматься разработкой, но его экспертная область включает и конфигурацию сети, и Kubernetes, и производительность с устойчивостью, и инструменты/системы, позволяющие его коллегам доставлять свой код в облако.

Благодаря революции DevOps в большинстве организаций не останется места для разработчиков без навыков системного администрирования и для системных администраторов без навыков разработки. Разграничение между этими двумя направлениями устарело и стремительно сводится на нет. Разработка и администрирование ПО — лишь два аспекта одной и той же деятельности.

Некоторые элементы останутся централизованными

Есть ли у DevOps предел? Или традиционные централизованные отделы IT и системного администрирования полностью исчезнут, превратившись в группу внутренних консультантов, которые обучают персонал и решают инфраструктурные задачи?

Мы считаем, что этого не произойдет. Многое по-прежнему лучше работает при централизованном управлении. Например, нет никакого смысла поддерживать параллельные способы обнаружения и обсуждения производственных инцидентов, системы ведения тикетов или инструментов развертывания для каждого отдельного приложения или обслуживающей команды. Если все начнут изобретать свой собственный велосипед, ничего хорошего не получится.

Планирование продуктивности разработчиков

Суть в том, что у самообслуживания есть свои пределы, и DevOps нацелен на повышение продуктивности разработчиков, а не на замедление их работы за счет дополнительных и ненужных обязанностей.

Действительно, огромная часть традиционного системного администрирования должна быть поручена другим командам, в основном тем, которые среди прочего занимаются развертыванием и реагированием на происшествия, связанные с кодом. Но, чтобы подобное стало возможным, вам нужна сильная центральная команда — для формирования и поддержки экосистемы DevOps, в рамках которой будут работать все остальные команды.

Вместо *системного администрирования* (operations) мы предпочитаем говорить о *планировании продуктивности разработчиков* (developer productivity engineering, или DPE). Команды DPE делают все необходимое для того, чтобы разработчики могли выполнять свои обязанности лучше и быстрее: управляют инфраструктурой, создают инструменты, решают проблемы.

DPE остается специализированным набором навыков, но такие специалисты могут сами переходить из одного отдела организации в другой, применяя свои знания там, где они нужны.

Мэтт Клейн, инженер компании Lyft, высказал мысль, что чистый DevOps имеет смысл для стартапов и мелких фирм, тогда как по мере роста организации проявляется естественная тенденция к тому, что эксперты в области инфраструктуры и надежности начинают тяготеть к централизованному стилю работы. Но он также отмечает, что такую централизованную команду нельзя масштабировать вечно.

> К моменту, когда организация, занимающаяся инженерной деятельностью, достигает ~75 человек, у нее уже почти наверняка есть центральная инфраструктурная команда, которая приступает к созданию инструментов общего характера, необходимых командам для построения конечных микросервисов. Но рано или поздно подобная команда утрачивает способность создавать и администрировать инфраструктуру, критически важную для успеха компании, но сохраняет при этом за собой обязанности отдела поддержки, помогая командам с административными задачами.
>
> *Мэтт Клейн*

На сегодняшний день не каждый разработчик может выступать специалистом по инфраструктуре. Точно так же одна команда инфраструктурных специалистов не может обслуживать разработчиков, число которых постоянно растет. Крупные

организации все еще испытывают необходимость в центральной инфраструктурной команде, но у каждой группы разработчиков и команды для построения конечных микросервисов есть место и для *инженеров по обеспечению безопасности информационных систем* (site reliability engineers, SRE). Они консультируют и наводят мосты между разработкой продукта и администрированием инфраструктуры.

Вы — будущее

Если вы читаете нашу книгу, это означает, что вскоре вы станете частью этого облачно-ориентированного будущего. В последующих главах мы поделимся всеми знаниями и навыками, которые могут понадобиться разработчику или системному администратору, имеющему дело с облачной инфраструктурой, контейнерами и Kubernetes.

Одни понятия покажутся вам знакомыми, а другие будут в новинку. Мы надеемся, что по прочтении книги вы будете более уверены в своей способности овладевать облачно-ориентированными навыками и доводить их до совершенства. Да, вам многое предстоит изучить, но нет ничего такого, с чем бы вы не справились. У вас все получится!

Теперь читайте дальше.

Резюме

Мы, как и следовало в начале, провели быстрый и краткий обзор DevOps. Надеемся, его оказалось достаточно для того, чтобы вы получили представление и о тех проблемах, с которыми справляются облака, контейнеры и Kubernetes, и о том, как они, скорее всего, изменят индустрию информационных технологий. Если эта тема вам уже была знакома, благодарим за терпение.

Прежде чем переходить к следующей главе, в которой вы познакомитесь с Kubernetes лицом к лицу, пройдемся по основным моментам.

- Облачные вычисления освобождают вас от накладных расходов, связанных с управлением собственным аппаратным обеспечением, и дают вам возможность создавать устойчивые, гибкие, масштабируемые распределенные системы.
- Наличие DevOps — это признание того, что разработка современного ПО не заканчивается доставкой кода. Данный подход завершает цикл обратной связи между теми, кто этот код пишет, и теми, кто его использует.

- DevOps также ставит код во главу угла и делает доступными проверенные методики разработки ПО в мире инфраструктуры и системного администрирования.
- Контейнеры позволяют развертывать и запускать ПО в виде небольших самодостаточных модулей. Это облегчает и удешевляет построение крупных разносторонних распределенных систем за счет объединения контейнеризированных микросервисов.
- Системы оркестрации берут на себя развертывание ваших контейнеров, планирование, масштабирование, подключение к сети и все те действия, которыми должен заниматься хороший системный администратор, — но делают это автоматизированным программируемым способом.
- Kubernetes является стандартом де-факто системы оркестрации контейнеров и уже сегодня готова к использованию в промышленных условиях.
- Термин *«облачно-ориентированный»* удобен для разговора об облачных контейнеризированных распределенных системах, состоящих из взаимодействующих микросервисов, которые, словно код, динамически управляются автоматизированной инфраструктурой.
- Несмотря на облачную революцию, владеть навыками системного администрирования и работы с инфраструктурой все еще нужно — они становятся еще более важными, чем раньше.
- Построение и обслуживание платформ и инструментов, которые делают DevOps возможным во всех остальных командах, все еще имеет смысл поручать отдельной центральной команде.

У чего нет будущего, так это у четкого разделения между разработчиками и системными администраторами. Теперь все это просто программное обеспечение, а мы с вами все — инженеры.

2 Первые шаги с Kubernetes

> Чтобы сделать что-то по-настоящему стоящее, мне надо не отступать и трястись при мыслях о холоде и опасности, а с радостью дерзать и идти вперед изо всех сил.
>
> *Ог Мандино*

Хватит теории, давайте начнем работать с Kubernetes и контейнерами. В этой главе вы создадите простое контейнеризированное приложение и развернете его в локальном кластере Kubernetes, запущенном на вашем компьютере. По ходу вы познакомитесь с некоторыми очень важными облачными технологиями и концепциями: Docker, Git, Go, реестром контейнеров и утилитой `kubectl`.

Эта глава интерактивная! Мы нередко будем просить вас что-либо установить на компьютер, ввести команды и запустить контейнеры — именно для того, чтобы вы могли следовать нашим примерам. Мы считаем, что так обучать гораздо эффективнее, чем просто объяснять на словах.

Запускаем наш первый контейнер

Как вы уже узнали из главы 1, контейнер является одной из ключевых концепций в облачно-ориентированной разработке. Основным инструментом для построения и выполнения контейнеров выступает Docker. В этом разделе мы будем использовать программу Docker Desktop, чтобы создать простое демонстрационное приложение, локально запустить и поместить его образ в реестр контейнеров.

Если вы уже хорошо знакомы с контейнерами, можете сразу переходить к разделу «Здравствуй, Kubernetes» на с. 60, с которого начинается настоящее веселье. Если же вам интересно, что такое контейнеры и как они работают, и вы, прежде

чем приступать к изучению Kubernetes, хотите немного с ними попрактиковаться, читайте дальше.

Установка Docker Desktop

Docker Desktop — это полноценная среда разработки для Mac и Windows, которую можно запускать на ноутбуке или настольном компьютере. Она включает в себя одноузловой кластер Kubernetes, с помощью которого вы можете тестировать свои приложения.

Установим среду Docker Desktop и воспользуемся ею для выполнения простого контейнеризированного приложения. Если у вас в системе уже есть Docker, пропускайте этот раздел и перелистывайте сразу к подразделу «Запуск образа контейнера» на с. 53.

Загрузите версию Docker Desktop Community Edition (hub.docker.com/search/?type=edition&offering=community), которая подходит для вашего компьютера, и затем следуйте инструкциям для своей платформы, чтобы установить и запустить Docker.

На сегодняшний день среда Docker Desktop недоступна для Linux, поэтому пользователям данной операционной системы необходимо установить Docker Engine (www.docker.com/products/container-runtime) и затем Minikube (см. раздел «Minikube» на с. 62).

Когда вы это сделаете, у вас должна появиться возможность открыть терминал и выполнить следующую команду:

```
docker version
Client:
 Version:      18.03.1-ce
 ...
```

Этот вывод в деталях может иметь некоторые различия в зависимости от вашей платформы, но если пакет Docker корректно установлен и запущен, вы должны увидеть нечто похожее. В системах Linux вместо этого вам, возможно, придется выполнить команду `sudo docker version`.

Что такое Docker

На самом деле Docker (docs.docker.com) — это сразу несколько разных, но связанных между собой инструментов: формат образов для контейнеров, библиотека *среды выполнения контейнеров*, управляющая их жизненным циклом, утилита командной строки для упаковки и запуска контейнеров, а также API для управления контей-

нерами. Подробности здесь нас заботить не должны, поскольку, несмотря на свою значимость, Docker является лишь одним из многих компонентов Kubernetes.

Запуск образа контейнера

Что именно представляет собой образ контейнера? Для наших целей технические подробности не очень-то и важны, но вы можете относиться к образу как к ZIP-файлу. Это единый двоичный файл с уникальным идентификатором, который хранит все, что необходимо для работы контейнера.

Неважно, как запускается ваш контейнер — напрямую с помощью Docker или в Kubernetes: достаточно лишь указать ID или URL его образа, и система сама его найдет, загрузит, распакует и запустит.

Мы написали небольшое демонстрационное приложение, которое будем использовать в этой книге для иллюстрации того, о чем идет речь. Вы можете загрузить и запустить его с помощью образа контейнера, который мы подготовили заблаговременно. Чтобы его опробовать, выполните следующую команду:

```
docker container run -p 9999:8888 --name hello cloudnatived/demo:hello
```

Пусть эта команда работает, а вы перейдите в свой браузер и откройте адрес http://localhost:9999/.

Вы должны увидеть приветственное сообщение:

```
Hello, 世界
```

Каждый раз, когда вы обратитесь по этому URL-адресу, наше демонстрационное приложение с готовностью вас поприветствует.

Вдоволь позабавившись, можете остановить контейнер, нажав Ctrl+C в своем терминале.

Демонстрационное приложение

Так как же это работает? Загрузим исходный код нашего демонстрационного приложения, которое выполняется в этом контейнере, и посмотрим.

Для этого нужно установить пакет Git[1]. Если вы не уверены в том, что он у вас есть, попробуйте следующую команду:

```
git version
git version 2.17.0
```

[1] Если вы не знакомы с Git, почитайте замечательную книгу: *Чакон С., Штрауб Б.* Git для профессионального программиста. — СПб.: Питер, 2019.

Если у вас не оказалось Git, следуйте инструкциям (git-scm.com/download) для своей платформы.

Закончив с установкой, выполните такую команду:

```
git clone https://github.com/cloudnativedevops/demo.git
Cloning into demo...
...
```

Рассмотрение исходного кода

Репозиторий Git содержит демонстрационное приложение, которое мы будем использовать на страницах этой книги. Для облегчения понимания того, что происходит на каждом из этапов, в репозитории находятся все версии приложения, причем в отдельных папках. Первая называется просто hello. Чтобы просмотреть исходный код, выполните следующую команду:

```
cd demo/hello
ls
Dockerfile  README.md
go.mod      main.go
```

Откройте файл `main.go` в том редакторе, который вам нравится (мы рекомендуем среду Visual Studio Code (https://code.visualstudio.com/), имеющую прекрасную поддержку разработки на Go, Docker и Kubernetes). Вы увидите такой исходный код:

```
package main

import (
        "fmt"
        "log"
        "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, 世界")
}

func main() {
        http.HandleFunc("/", handler)
        log.Fatal(http.ListenAndServe(":8888", nil))
}
```

Введение в Go

Наше демонстрационное приложение написано на языке программирования Go.

Go — это современный язык программирования, представленный компанией Google в 2009 году, который ставит на первое место простоту, безопасность и понятность

кода. Он предназначен для создания крупномасштабных конкурентных приложений с акцентом на сетевых сервисах. Кроме того, на нем довольно интересно писать[1].

Сама платформа Kubernetes написана на Go, так же как и Docker, Terraform и многие другие популярные проекты с открытым исходным кодом. Поэтому данный язык — хороший выбор для разработки облачно-ориентированных программ.

Как работает демонстрационное приложение

Как видите, демонстрационное приложение довольно простое, несмотря на то что оно реализует HTTP-сервер (Go поставляется вместе с мощной стандартной библиотекой). В его основе лежит функция под названием `handler`:

```
func handler(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello, 世界")
}
```

Как можно догадаться из названия, функция обрабатывает HTTP-запросы, которые передаются ей в виде аргумента (хотя пока ничего с ними не делает).

У HTTP-сервера также должно быть какое-то средство для отправки ответов клиенту. Объект `http.ResponseWriter` позволяет нашей функции послать пользователю сообщение, которое отобразится в его браузере: в данном случае это просто строка `Hello, 世界`.

Первое действие программы на любом языке традиционно `Hello, world`. Но, поскольку Go имеет встроенную поддержку Unicode (международный стандарт для представления текста), программы Go часто печатают `Hello, 世界` — просто чтобы покрасоваться. Если вы не знаете китайского, не беда: Go знает!

Остальная часть программы занимается регистрацией функции `handler` в качестве обработчика HTTP-запросов и запуском HTTP-сервера для прослушивания и обслуживания порта 8888.

Вот и все приложение! Пока оно не делает ничего особенного, но постепенно мы будем добавлять в него разные возможности.

Построение контейнера

Вы уже знаете, что образ контейнера — это единый файл, содержащий все, что этому контейнеру нужно для работы. Но как создать такой образ? Что ж, для этого используется команда `docker image build`, которая принимает в качестве входных

[1] Если вы достаточно опытный программист, но не знакомы с Go, руководство The Go Programming Language (Addison-Wesley) от Алана Донована и Брайана Кернигана окажется для вас бесценным (https://www.gopl.io/).

данных специальный текстовый файл под названием `Dockerfile`. Dockerfile точно описывает, что именно должно находиться в образе контейнера.

Одним из ключевых преимуществ контейнеров является возможность построения новых образов на основе существующих. Например, вы можете взять образ контейнера с полноценной операционной системой Ubuntu внутри, добавить к нему один файл и в итоге получить новый образ.

В целом у Dockerfile есть инструкции для преобразования начального (так называемого *базового*) образа тем или иным способом и сохранения результата в виде нового образа.

Что собой представляет Dockerfile

Рассмотрим файл `Dockerfile` для нашего демонстрационного приложения (он находится в подкаталоге `hello` внутри репозитория приложения):

```
FROM golang:1.11-alpine AS build

WORKDIR /src/
COPY main.go go.* /src/
RUN CGO_ENABLED=0 go build -o /bin/demo

FROM scratch
COPY --from=build /bin/demo /bin/demo
ENTRYPOINT ["/bin/demo"]
```

Пока нам неважно, как именно это работает, однако здесь используется довольно стандартный для контейнеров Go процесс под названием *многоступенчатая сборка*. Первая ступень — официальный образ контейнера `golang`, который представляет собой просто операционную систему (в данном случае Alpine Linux) с установленной средой языка Go. Он выполняет команду `go build`, чтобы скомпилировать уже знакомый нам файл `main.go`.

В результате получается исполняемый двоичный файл с именем `demo`. На втором этапе берется абсолютно пустой образ контейнера под названием `scratch` (от англ. *from scratch* — «с нуля») и в него копируется файл `demo`.

Минимальные образы контейнеров

Зачем нужен второй этап сборки? Действительно, среда языка Go и остальная часть Alpine Linux используются лишь для того, чтобы *собрать* программу. А для ее запуска достаточно только двоичного файла `demo`, поэтому Dockerfile и создает новый

контейнер `scratch`, чтобы поместить его туда. Итоговый образ получается очень маленьким (около 6 Мбайт), и его уже можно развернуть в реальных условиях.

Без второй ступени у вас бы получился образ контейнера размером примерно 350 Мбайт, 98 % которого не нужны и никогда не будут выполняться. Чем меньше образ контейнера, тем быстрее его можно загрузить в реестр и обратно и тем быстрее он будет запускаться.

Минимальные контейнеры также уменьшают *поверхность атаки*, улучшая тем самым безопасность. Чем меньше программ в вашем контейнере, тем меньше потенциальных уязвимостей.

Поскольку Go является компилируемым языком, способным генерировать полноценные исполняемые файлы, он идеально подходит для написания минимальных контейнеров (scratch). Для сравнения, официальный образ контейнера для Ruby занимает 1,5 Гбайт, что примерно в 250 раз больше, чем образ для Go. И это не считая самой программы на Ruby!

Выполнение команды docker image build

Как мы уже видели, Dockerfile содержит инструкции для команды `docker image build`, которые превращают наш исходный код на Go в исполняемый контейнер. Попробуйте это сделать самостоятельно. Выполните следующую команду в каталоге `hello`:

```
docker image build -t myhello .
Sending build context to Docker daemon 4.096kB
Step 1/7 : FROM golang:1.11-alpine AS build
...

Successfully built eeb7d1c2e2b7
Successfully tagged myhello:latest
```

Поздравляем, вы только что собрали свой первый контейнер! По выводу видно, что Docker выполняет каждое действие в Dockerfile по очереди и для вновь сформированного контейнера. В итоге получается готовый к использованию образ.

Выбор имен для ваших образов

Когда вы собираете образ, он по умолчанию получает шестнадцатеричный идентификатор, с помощью которого позже на образ можно будет ссылаться — например, чтобы запустить его. Такие идентификаторы не очень хорошо запоминаются,

и вводить их непросто. Поэтому Docker позволяет давать образам понятные для человека названия: для этого у команды `docker image build` предусмотрен переключатель `-t`. В предыдущем примере вы назвали свой образ `myhello`, поэтому теперь можете запускать его с помощью этого имени.

Посмотрим, как это у вас получится:

```
docker container run -p 9999:8888 myhello
```

В результате у вас должна заработать ваша собственная копия демонстрационного приложения. Чтобы в этом убедиться, откройте тот же, что и прежде, URL-адрес (http://localhost:9999/).

Вы должны увидеть `Hello, 世界`. Закончив работу с этим образом, нажмите Ctrl+C, чтобы остановить команду `docker container run`.

УПРАЖНЕНИЕ

Если хотите немного поразвлечься, откройте файл `main.go` в демонстрационном приложении и измените приветствие так, чтобы оно выводилось на вашем любимом языке (или вообще поменяйте его на то, что вам нравится). Пересоберите и запустите контейнер, чтобы проверить, все ли работает.

Поздравляем, теперь вы программист на Go! Но не останавливайтесь на этом: чтобы узнать больше, пройдите интерактивный ознакомительный курс (tour.golang.org/welcome/1).

Перенаправление портов

Программы, запущенные в контейнере, изолированы от других программ, работающих на том же компьютере. Это означает, что у них нет прямого доступа к таким ресурсам, как сетевые порты.

Демонстрационное приложение ожидает соединений на порте 8888, но это приватный порт самого *контейнера*, а не вашего компьютера. Чтобы подключиться к порту контейнера 8888, необходимо *перенаправить* порт на локальном компьютере на этот порт контейнера. Это может быть любой порт, включая 8888, но вместо этого мы выберем 9999, чтобы вам было понятно, какой из них принадлежит вам, а какой — контейнеру.

Чтобы заставить Docker перенаправить порт, можно использовать переключатель `-p` точно так же, как вы это делали ранее в подразделе «Запуск образа контейнера» на с. 53:

```
docker container run -p HOST_PORT:CONTAINER_PORT ...
```

Когда контейнер запустится, любой запрос к `HOST_PORT` на локальном компьютере будет автоматически перенаправлен к порту контейнера `CONTAINER_PORT`. И это позволит подключиться к приложению с помощью вашего браузера.

Реестры контейнеров

В подразделе «Запуск образа контейнера» на с. 53 вам удалось запустить образ, просто указав его название, — Docker произвел загрузку автоматически.

Разумно будет поинтересоваться, откуда образ был загружен. Docker вполне можно использовать для сборки и запуска локальных образов, но куда более полезной является возможность *загружать их в реестр контейнеров и выгружать из него*. Этот реестр позволяет хранить и извлекать образы по уникальному имени (такому как `cloudnatived/demo:hello`).

Команда `docker container run` по умолчанию использует реестр Docker Hub, но вы можете указать любой другой или подготовить свой собственный.

Пока остановимся на Docker Hub. Оттуда вы можете загрузить и использовать любой публичный образ контейнера, но, чтобы отправлять туда собственные образы, вам понадобится учетная запись (которую называют *Docker ID*). Чтобы создать Docker ID, следуйте инструкциям по адресу hub.docker.com.

Аутентификация в реестре

Следующим шагом после получения Docker ID будет подключение вашего локального сервиса Docker (часто его называют «демон Docker») к Docker Hub с использованием ваших ID и пароля:

```
docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: YOUR_DOCKER_ID
Password: YOUR_DOCKER_PASSWORD
Login Succeeded
```

Выбор имени для вашего образа и загрузка его в реестр

Чтобы разместить свой локальный образ в реестре, вам нужно назвать его в таком формате: *YOUR_DOCKER_ID*/myhello.

Для создания этого имени не обязательно пересобирать весь образ — достаточно выполнить следующую команду:

```
docker image tag myhello YOUR_DOCKER_ID/myhello
```

Это делается для того, чтобы при загрузке образа в реестр Docker знал, в какой учетной записи его нужно хранить.

Загрузите свой образ в Docker Hub с помощью такой команды:

```
docker image push YOUR_DOCKER_ID/myhello
The push refers to repository [docker.io/YOUR_DOCKER_ID/myhello]
b2c591f16c33: Pushed
latest: digest:
        sha256:7ac57776e2df70d62d7285124fbff039c9152d1bdfb36c75b5933057cefe4fc7
size: 528
```

Запуск вашего образа

Поздравляем! Теперь ваш образ контейнера можно запускать где угодно (при условии наличия доступа к Интернету), используя такую команду:

```
docker container run -p 9999:8888 YOUR_DOCKER_ID/myhello
```

Здравствуй, Kubernetes

Итак, вы собрали и загрузили в реестр свой первый образ контейнера. Его можно запустить с помощью команды `docker container run`, но это будет не так увлекательно. Давайте сделаем что-то более интересное: запустим образ в Kubernetes.

Получить кластер Kubernetes можно множеством способов, и мы подробно рассмотрим некоторые из них в главе 3. Если у вас уже есть доступ к такому кластеру — отлично; если хотите, можете использовать его для выполнения остальных примеров в этой главе.

В противном случае не волнуйтесь. Docker Desktop имеет поддержку Kubernetes (а пользователям Linux следует перейти к разделу «Minikube» на с. 62). Чтобы

ее включить, откройте панель настроек Docker Desktop, перейдите на вкладку Kubernetes и установите флажок Enable Kubernetes (Включить Kubernetes), как показано на рис. 2.1.

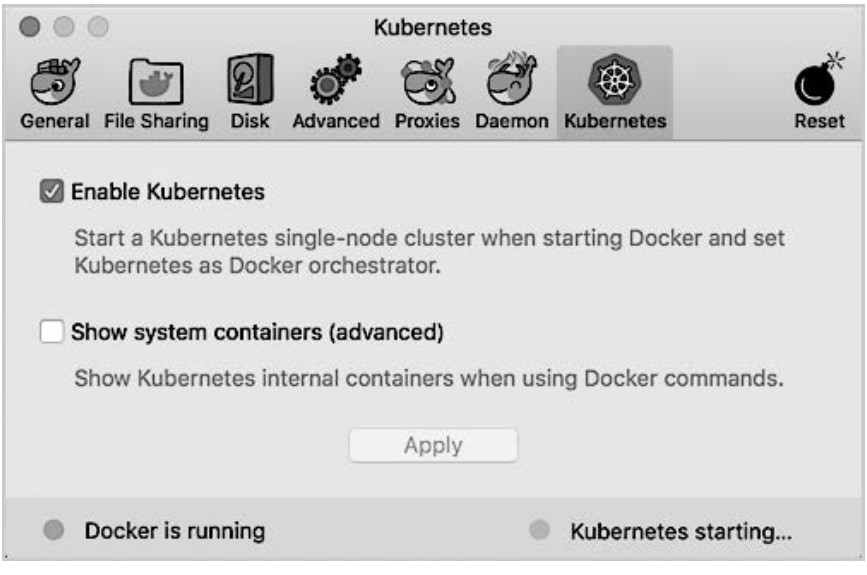

Рис. 2.1. Включение поддержки Kubernetes в Docker Desktop

На установку и запуск Kubernetes уйдет несколько минут. По окончании вы будете готовы к запуску демонстрационного приложения!

Запуск демонстрационного приложения

Для начала запустим демонстрационное приложение, которое вы собрали ранее. Откройте терминал и введите команду `kubectl` со следующими аргументами:

```
kubectl run demo --image=YOUR_DOCKER_ID/myhello --port=9999 --labels app=demo
deployment.apps "demo" created
```

Пока не обращайте внимания на детали: это, в сущности, эквивалент команды `docker container run`, с помощью которой в этой главе вы уже запускали демонстрационный контейнер, только для Kubernetes. Если свой образ вы еще не собрали, можете воспользоваться нашим: `--image=cloudnatived/demo:hello`.

Напомним, что вам следует перенаправить порт 9999 вашего локального компьютера к порту 8888 контейнера, чтобы к нему можно было подключаться с помощью

веб-браузера. То же самое необходимо сделать здесь, используя команду `kubectl port-forward`:

```
kubectl port-forward deploy/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

Пусть эта команда продолжает работать, а вы откройте новый терминал.

Перейдя по адресу http://localhost:9999/ в своем браузере, вы увидите сообщение `Hello, 世界`.

На то, чтобы запустить контейнер и сделать приложение доступным, может уйти несколько секунд. Если это не будет выполнено через полминуты или около того, попробуйте следующую команду:

```
kubectl get pods --selector app=demo
NAME                    READY     STATUS    RESTARTS   AGE
demo-54df94b7b7-qgtc6   1/1       Running   0          9m
```

Когда контейнер запустится и вы подключитесь к нему с помощью своего браузера, в терминале появится такое сообщение:

```
Handling connection for 9999
```

Если контейнер не запускается

Если в поле `STATUS` не указано `Running`, вероятно, у вас возникли проблемы. Например, если вы видите статус `ErrImagePull` или `ImagePullBackoff`, это означает, что кластер Kubernetes не смог найти и загрузить указанный вами образ. Возможно, вы допустили опечатку в имени образа — проверьте свою команду `kubectl run`.

Если же статус `ContainerCreating`, значит, все хорошо и просто Kubernetes все еще загружает или запускает ваш образ. Подождите несколько секунд и проверьте снова.

Minikube

Если вы не хотите или не можете использовать поддержку Kubernetes в Docker Desktop, у вас есть альтернатива: всеми любимая система Minikube. Как и Docker Desktop, она предоставляет одноузловой кластер Kubernetes, который работает на вашем собственном компьютере (на самом деле он запущен внутри виртуальной машины, но это неважно).

Чтобы установить Minikube, следуйте инструкциям по адресу kubernetes.io/docs/tasks/tools/install-minikube.

Резюме

Если вам, как и нам, быстро надоели многословные оды в адрес Kubernetes, то надеемся, знакомство с некоторыми практическими задачами из этой главы пришлось вам по душе. Если вы уже являетесь опытным пользователем Docker или Kubernetes, возможно, вы простите нам этот «курс переподготовки». Прежде чем приступать к углубленным темам, мы хотим убедиться в том, что все читатели подготовлены к сборке и запуску контейнеров на простейшем уровне и что у вас есть среда Kubernetes, с которой вы можете экспериментировать.

Кратко перечислим основные итоги этой главы.

- Все примеры исходного кода (и многое другое) доступны в демонстрационном репозитории (github.com/cloudnativedevops/demo), который дополняет эту книгу.
- Утилита Docker позволяет собирать контейнеры локально, загружать их в реестр (такой как Docker Hub) и выгружать из него, и запускать их образы на вашем собственном компьютере.
- Образ контейнера полностью определяется текстовым файлом `Dockerfile`, который содержит инструкции о том, как собирать контейнер.
- Docker Desktop позволяет запускать на вашем компьютере небольшие (одноузловые) кластеры Kubernetes, которые тем не менее способны выполнять любые контейнеризированные приложения. Альтернативой является Minikube.
- Утилита `kubectl` служит основным способом взаимодействия с кластером Kubernetes. Ее можно использовать либо *императивно* (например, для запуска публичных образов контейнеров с явным выделением необходимых ресурсов Kubernetes), либо *декларативно*, чтобы применять конфигурацию Kubernetes в виде манифестов формата YAML.

3 Размещение Kubernetes

Недоумение — начало познания.

Халиль Джебран

Kubernetes — операционная система облачного мира, которая предоставляет надежную и масштабируемую платформу для выполнения контейнеризированных рабочих заданий. Но как следует запускать ее саму? Нужно ли хранить ее на своем компьютере? Или на облачных серверах? На физических серверах? А может, лучше воспользоваться управляемым сервисом или платформой, основанной на Kubernetes, но с дополнительными инструментами для организации рабочих процессов, панелями управления и веб-интерфейсами?

Довольно много вопросов для одной главы, но мы попытаемся на них ответить.

Стоит отметить, что здесь мы не будем слишком углубляться в технические подробности работы с самой платформой Kubernetes, такие как сборка, тонкая настройка и отладка кластеров. С этим вам может помочь множество замечательных ресурсов, среди которых мы особенно рекомендуем книгу *Managing Kubernetes: Operating Kubernetes Clusters in the Real World* (shop.oreilly.com/product/0636920146667.do) (O'Reilly) одного из основателей Kubernetes Брендана Бернса.

Вместо этого мы сосредоточимся на том, чтобы помочь вам понять базовую архитектуру кластера и дать всю информацию, необходимую для того, чтобы вы могли решить, как именно запускать Kubernetes: выделим преимущества и недостатки управляемых сервисов, а также рассмотрим некоторых популярных поставщиков.

Если же вы хотите запустить собственный кластер, воспользуйтесь инструментами для установки из числа лучших, которые мы перечислим далее, чтобы помочь вам с настройкой и управлением Kubernetes.

Архитектура кластера

Вы уже знаете, что Kubernetes объединяет разные серверы в *кластер*. Но что такое кластер и как он работает? В контексте этой книги технические подробности неважны, однако вы должны понимать, из каких основных компонентов состоит Kubernetes и как они совмещаются друг с другом; это нужно для понимания того, какой выбор вам доступен при построении или покупке кластеров Kubernetes.

Управляющий уровень

«Мозг» кластера называется *управляющим уровнем*. Он берет на себя все задачи, необходимые для того, чтобы платформа Kubernetes выполняла свою работу: планирование контейнеров, управление сервисами, обслуживание API-запросов и т. д. (рис. 3.1).

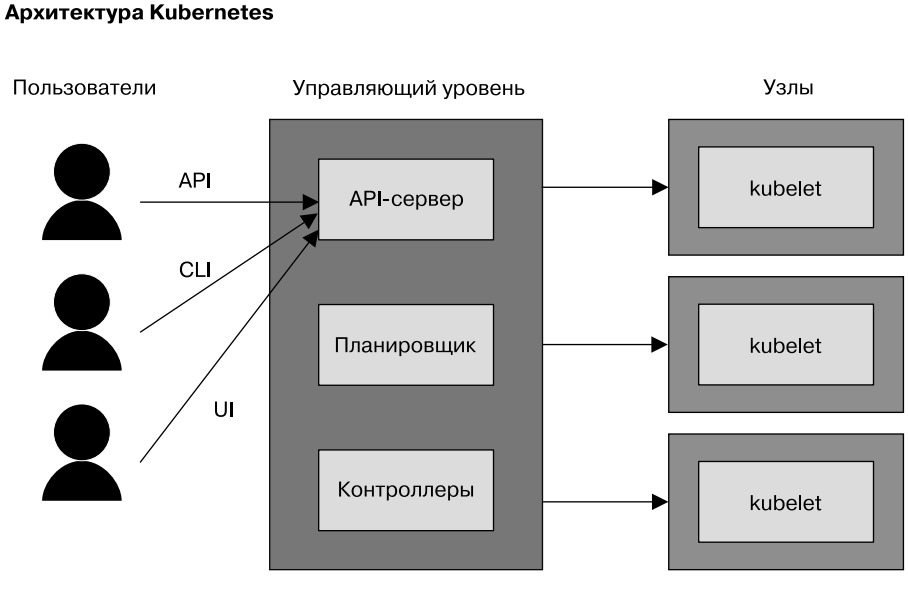

Рис. 3.1. Как работает кластер Kubernetes

Управляющий уровень на самом деле состоит из нескольких компонентов.

- `kube-apiserver` — это внешний сервер для управляющего уровня, который обрабатывает API-запросы.
- `etcd` — база данных, в которой Kubernetes хранит всю информацию о существующих узлах, ресурсах кластера и т. д.
- `kube-scheduler` определяет, где будут запущены свежесозданные pod-оболочки.
- `kube-controller-manager` отвечает за запуск контроллеров ресурсов, таких как развертывания.
- `cloud-controller-manager` взаимодействует с облачным провайдером (в облачных кластерах), управляя такими ресурсами, как балансировщики нагрузки и дисковые тома.

Участники кластера, которые выполняют компоненты управляющего уровня, называются *ведущими узлами*.

Компоненты узла

Участники кластера, которые выполняют пользовательские рабочие задания, называются *рабочими узлами* (рис. 3.2).

Каждый рабочий узел в кластере Kubernetes отвечает за следующие компоненты.

- `kubelet` отвечает за управление средой выполнения контейнера, в которой запускаются рабочие задания, запланированные для узла, а также за мониторинг их состояния.
- `kube-proxy` занимается сетевой магией, которая распределяет запросы между pod-оболочками на разных узлах, а также между pod-оболочками и Интернетом.
- *Среда выполнения контейнеров* запускает и останавливает контейнеры, а также отвечает за их взаимодействие. Обычно это Docker, хотя Kubernetes поддерживает и другие среды выполнения контейнеров, такие как `rkt` и CRI-O.

Кроме как выполнением разных программных компонентов, ведущие и рабочие узлы принципиально ничем не отличаются. Хотя ведущий узел обычно не выполняет пользовательские рабочие задания, исключение составляют очень маленькие кластеры (такие как Docker Desktop или Minikube).

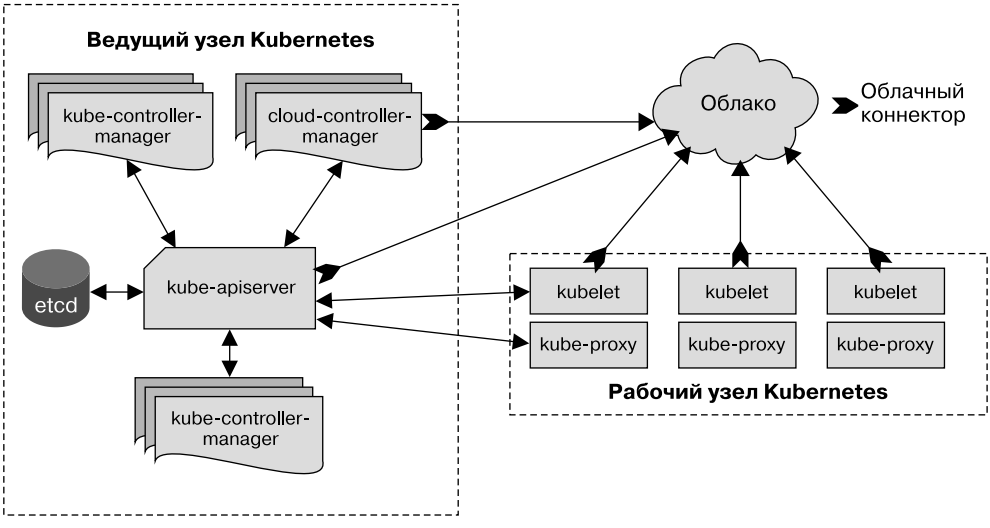

Рис. 3.2. Взаимоотношение компонентов Kubernetes

Высокая доступность

У правильно сконфигурированного управляющего уровня Kubernetes есть несколько ведущих узлов, что делает его *высокодоступным*. Иначе говоря, если отдельный ведущий узел выйдет из строя или какой-то из его компонентов прекратит работу, кластер продолжит функционировать в обычном режиме. Высокодоступный управляющий уровень также способен справиться с ситуациями, когда ведущие узлы работают должным образом, но некоторые из них не могут связаться с остальными из-за сетевых неполадок (известных как *сетевое разделение*).

База данных `etcd` реплицируется между несколькими узлами и может пережить отказ отдельных копий при условии наличия кворума из более чем половины реплик `etcd`.

Если все это сконфигурировано как следует, управляющий уровень может пережить перегрузку или временные неисправности на отдельных ведущих узлах.

Отказ управляющего уровня

Повреждение управляющего уровня не обязательно приводит к отказу вашего приложения, хотя странное и ошибочное поведение вполне возможно.

Например, если вы остановите все ведущие узлы в своем кластере, pod-оболочки на рабочих узлах продолжат функционировать — по крайней мере какое-то время. Однако вы не сможете развертывать новые контейнеры или менять какие-либо ресурсы Kubernetes, а такие контроллеры, как развертывания, перестанут действовать.

Вот поэтому высокая доступность управляющего уровня крайне важна для правильной работы кластера. Вам необходимо запастись достаточным количеством ведущих узлов, чтобы кластер мог поддерживать *кворум*, даже если какой-то из узлов откажет. Для промышленных кластеров реалистичным минимумом является три узла (см. пункт «Минимальный кластер» на с. 138).

Отказ рабочего узла

Отказ любого рабочего узла, напротив, не влечет за собой никаких существенных последствий: Kubernetes обнаружит сбой и перераспределит pod-оболочки этого узла. Главное, чтобы работал управляющий уровень.

Одновременный отказ большого количества узлов может привести к тому, что у кластера не хватит ресурсов для выполнения всех рабочих заданий, которые вам нужны. К счастью, это случается нечасто, и в таких ситуациях Kubernetes постарается поддерживать работу как можно большего числа ваших pod-оболочек, пока вы не замените потерянные узлы.

Тем не менее стоит помнить о том, что чем меньше у вас рабочих узлов, тем большую долю производительности кластера каждый из них представляет. Вы должны исходить из того, что отдельный узел может выйти из строя в любой момент, особенно в облаке, и что два одновременных сбоя не являются чем-то невообразимым.

Редким, но вполне возможным происшествием можно считать потерю целой *зоны доступности*. Поставщики облачных сервисов, такие как AWS и Google, в каждом регионе обеспечивают несколько зон доступности (зона примерно соответствует отдельному вычислительному центру). Поэтому было бы неплохо распределить рабочие узлы между двумя или даже тремя зонами, а не размещать их в одной и той же.

Доверяй, но проверяй

Благодаря высокой доступности ваш кластер должен пережить потерю одного ведущего или нескольких рабочих узлов, но лучше это *перепроверить*. Попробуйте перезагрузить рабочий узел во время запланированного периода обслуживания или в моменты отсутствия пиковых нагрузок и посмотрите, что получится (надеемся, что ничего или же ничего такого, что было бы заметно пользователям ваших приложений).

Для более серьезной проверки перезагрузите один из ведущих узлов (управляемые сервисы вроде Google Kubernetes Engine, о которых мы поговорим позже в этой главе, по очевидным причинам не позволяют этого сделать). Кластер промышленного уровня должен пережить перезагрузку без каких-либо проблем.

Стоимость самостоятельного размещения Kubernetes

Если вы собираетесь выполнять реальные рабочие задания в Kubernetes, самый важный выбор, который вам придется сделать, таков: *купить или построить*? Стоит ли запускать свои собственные кластеры? Или лучше заплатить, чтобы кто-то другой их запускал? Давайте рассмотрим некоторые варианты.

Самым простым выбором из всех является самостоятельное размещение Kubernetes. Под *самостоятельным размещением* мы имеем в виду, что вы (лично или вместе с какой-то командой в вашей организации) устанавливаете и конфигурируете Kubernetes на компьютерах, находящихся под вашим контролем — точно так же, как делаете это с любым другим ПО, которое используете: Redis, PostgreSQL, Nginx и т. д.

Этот вариант дает вам максимальные гибкость и контроль: вы сами можете выбрать версию Kubernetes, то, какие параметры и возможности следует включить, когда обновлять кластеры и нужно ли это делать и т. д. Но у подобного подхода есть существенные недостатки, в чем вы сможете убедиться в следующем разделе.

Это сложнее, чем вам кажется

Вариант с самостоятельным размещением требует максимума ресурсов, таких как люди и их навыки, а также время на проектирование, обслуживание и отладку. Подготовка рабочего кластера Kubernetes сама по себе довольно проста, но это далеко не то же самое, что получение кластера, готового к промышленным нагрузкам. Как минимум вам следует рассмотреть следующие вопросы.

- Является ли управляющий уровень высокодоступным? То есть должен ли ваш кластер продолжать работу, если его ведущий узел выйдет из строя или перестанет отвечать? Сможете ли вы при этом развертывать или обновлять приложения? Останутся ли ваши запущенные задания отказоустойчивыми без управляющего уровня (см. подраздел «Высокая доступность» на с. 67)?
- Является ли высокодоступным ваш пул рабочих узлов? То есть не прекратят ли выполнение ваши рабочие задания в случае, если из-за перебоев в работе выйдет из строя несколько рабочих узлов или даже целая зона доступности облака?

Сможет ли ваш кластер автоматически выделить новые узлы, чтобы восстановить свою работоспособность, или же это потребует ручного вмешательства?

- Настроен ли ваш кластер *безопасным* образом? Используют ли его внутренние компоненты шифрование TLS и доверенные сертификаты для общения между собой? Назначены ли пользователям и приложениям минимальные права и полномочия для работы в кластере? Установлены ли для контейнера безопасные параметры по умолчанию? Имеют ли узлы излишний доступ к компонентам управляющего уровня? Установлены ли надлежащие контроль и аутентификация для доступа к внутренней базе данных `etcd`?
- Все ли сервисы в вашем кластере защищены? Если они доступны из Интернета, предусмотрены ли для них надлежащие аутентификация и авторизация? Является ли доступ к API кластера строго ограниченным?
- *Соответствует* ли ваш кластер установленным требованиям? Отвечает ли он стандартам для кластеров Kubernetes, определенным организацией Cloud Native Computing Foundation (см. раздел «Проверка на соответствие» на с. 146)?
- Полностью ли поведение узлов вашего кластера *определяется конфигурацией*, или, может быть, они настраиваются с помощью скриптов командной оболочки и затем живут своей жизнью? Операционная система и ядро на каждом узле требуют обновления, применения патчей безопасности и т. д.
- Проводится ли правильное резервное копирование данных вашего кластера, включая любое постоянное хранилище? Каков ваш процесс восстановления? Как часто вы тестируете восстановление?
- Как вы обслуживаете рабочий кластер после его настройки? Как выделяете новые узлы? Как выкатываются обновления Kubernetes и изменения конфигурации для имеющихся узлов? Как происходит масштабирование в ответ на нагрузку? Как вы следите за соблюдением политик?

Писатель и проектировщик распределенных систем Синди Сридхаран подсчитала (twitter.com/copyconstruct/status/1020880388464377856), что на зарплаты инженеров, которые проводят подготовку и запуск промышленной конфигурации Kubernetes с нуля, уходит примерно миллион долларов («и даже его может не хватить»). Эта цифра должна дать пищу для размышлений любому техническому руководителю, который подумывает о самостоятельном размещении Kubernetes.

Начальная настройка — это далеко не все

Имейте в виду, что учитывать эти факторы вам придется не только во время подготовки первого кластера, но и на протяжении жизненных циклов всех ваших кластеров. При внесении изменений или обновлении инфраструктуры Kubernetes

стоит подумать о том, какие последствия это будет иметь с точки зрения высокой доступности, безопасности и т. д.

Вам потребуется мониторинг, чтобы быть уверенными в том, что узлы кластера и все компоненты Kubernetes работают должным образом. Также нужно предусмотреть систему оповещений, чтобы сотрудники могли быть уведомлены обо всех проблемах в любое время суток.

Kubernetes по-прежнему быстро развивается: постоянно появляются новые возможности и обновления. Вам придется поддерживать свой кластер в актуальном состоянии и иметь представление о том, как эти изменения влияют на имеющуюся конфигурацию. Возможно, вам понадобится заново выделить свой кластер, чтобы иметь возможность пользоваться всеми преимуществами самых новых функций Kubernetes.

Кроме того, недостаточно сконфигурировать кластер правильным образом, прочитав несколько книг или статей, и затем оставить его как есть. Вам придется регулярно проверять конфигурацию — например, убрав ведущий узел и убедившись, что все по-прежнему работает.

В этом вам могут помочь инструменты для автоматического тестирования устойчивости, такие как Chaos Monkey от Netflix. Они позволяют время от времени случайным образом убирать узлы, pod-оболочки и сетевые соединения. В зависимости от надежности вашего облачного провайдера использование Chaos Monkey может оказаться излишним, так как реальные сбои тоже служат проверкой устойчивости вашего кластера и тех сервисов, которые в нем выполняются (см. раздел «Хаотическое тестирование» на с. 152).

Инструменты не сделают за вас всю работу

Существуют инструменты — множество инструментов, — которые могут помочь с подготовкой и конфигурацией кластеров Kubernetes. Большинство из них обещают максимальную простоту, отсутствие каких-либо усилий с вашей стороны и мгновенные решения. Печально, но факт, что подавляющее большинство этих инструментов, на наш взгляд, решают только простые проблемы, а сложные игнорируют.

А мощные, гибкие коммерческие средства корпоративного уровня обычно очень дорогие или же недоступны для общего пользования, поскольку продажа управляемого сервиса выгоднее продажи инструментов управления кластером общего назначения.

Kubernetes — это сложно

Несмотря на широко распространенное мнение о том, что в подготовке и управлении нет ничего сложного, в действительности *Kubernetes — это сложно*. Система выглядит на удивление простой, если учитывать ее возможности, но при этом ей приходится справляться с крайне трудными ситуациями, что приводит к усложнению ПО.

Необходимо понимать, что для того, чтобы научиться как следует управлять собственными кластерами и заниматься этим изо дня в день и из месяца в месяц, потребуется много времени и сил. Мы не собираемся отговаривать вас от использования Kubernetes, просто хотим, чтобы вы четко понимали, что именно нужно для самостоятельного управления системой. Это поможет вам принять взвешенное решение, основанное на сравнении использования управляемых сервисов и размещения Kubernetes внутри компании с точки зрения стоимости и других преимуществ.

Накладные расходы на администрирование

Если вы работаете в крупной организации, обладающей достаточными ресурсами для отдельной команды по управлению кластерами Kubernetes, для вас это, может, и не будет большой проблемой. Но для мелких и средних компаний или даже стартапов с парой инженеров накладные расходы на администрирование собственных кластеров Kubernetes могут оказаться непосильными.

Учитывая ограничения в бюджете и количестве сотрудников, которых можно задействовать в системном администрировании, какую часть ресурсов вы хотите потратить исключительно на обслуживание Kubernetes? Не лучше ли будет направить их на поддержку рабочих заданий вашего бизнеса? Является ли обслуживание Kubernetes силами ваших сотрудников более выгодным, чем использование управляемых сервисов?

Начните с управляемых сервисов

Возможно, вы немного удивляетесь тому, что в книге о Kubernetes вам советуют не запускать эту систему! По крайней мере, не советуют делать это самостоятельно. По причинам, изложенным в предыдущих разделах, мы считаем, что использование управляемых сервисов экономически выгоднее, чем локальное размещение Kubernetes. Если вам не нужно делать ничего странного и экспериментального,

что не поддерживается ни одним из поставщиков, серьезных причин для выбора самостоятельного размещения практически нет.

Исходя из нашего опыта и опыта многих людей, с которыми мы беседовали в процессе написания этой книги, использование управляемого сервиса, вне всяких сомнений, является лучшим способом работы с Kubernetes.

Если вы пытаетесь решить, подходит ли вам платформа Kubernetes как таковая, управляемый сервис позволит ее опробовать. Вы можете получить полностью рабочий, защищенный, высокодоступный кластер промышленного уровня всего за пару минут и несколько долларов в день (большинство облачных провайдеров предлагают и бесплатную версию, которая позволит вам работать с Kubernetes неделями или месяцами, не внося никакой оплаты). Даже если по окончании пробного периода вы решите, что вам лучше запускать собственный кластер, управляемый сервис послужит примером того, как это делается.

А если вы уже сами поэкспериментировали с Kubernetes, вас порадует то, насколько все упрощается благодаря управляемым сервисам. Скорее всего, вы не сами строили свой дом — зачем тогда создавать собственный кластер, если это можно доверить кому-то другому, еще и сэкономив деньги и получив более качественный результат?

В следующем разделе мы перечислим некоторые из наиболее популярных сервисов по управлению Kubernetes, выскажем о них свое мнение и посоветуем наш любимый вариант. Если вы по-прежнему будете сомневаться, во второй части этой главы мы рассмотрим установщики Kubernetes, с помощью которых можно построить свои собственные кластеры (см. раздел «Установщики Kubernetes» на с. 78).

Здесь следует сказать, что ни один из авторов книги не связан ни с каким облачным провайдером или коммерческим поставщиком Kubernetes. Никто нам не платит за рекомендацию отдельных продуктов или сервисов. Мы высказываем наше собственное мнение, основанное на личном опыте и взглядах сотен пользователей Kubernetes, с которыми мы разговаривали при написании этой книги.

Естественно, в мире Kubernetes все движется очень быстро, а рынок управляемых сервисов испытывает особенно жесткую конкуренцию. Возможности и сервисы, описанные в книге, будут быстро меняться. Представленный здесь список не является полным, однако мы старались включить в него сервисы, которые, как нам кажется, можно назвать лучшими, наиболее распространенными или важными по какой-либо другой причине.

Управляемые сервисы Kubernetes

Управляемые сервисы Kubernetes избавляют вас почти от всех накладных расходов, связанных с системным администрированием, подготовкой и работой кластеров Kubernetes, в особенности управляющего уровня. Управляемый сервис, в сущности, означает, что вы платите кому-то другому (например, Google) за обслуживание кластера.

Google Kubernetes Engine (GKE)

Как и следовало ожидать от авторов этой системы, компания Google предлагает полностью управляемый сервис Kubernetes (cloud.google.com/kubernetes-engine), который полноценно интегрирован с Google Cloud Platform. Чтобы создать кластер, просто выберите количество рабочих узлов и нажмите кнопку в веб-консоли GCP или воспользуйтесь инструментом Deployment Manager. В течение нескольких минут ваш кластер будет готов к работе.

Google обеспечивает мониторинг и замену неисправных узлов, автоматическое применение патчей безопасности и высокую доступность для управляющего уровня и `etcd`. Вы можете сделать так, чтобы ваши узлы самостоятельно обновлялись до последней версии Kubernetes в заданный вами период обслуживания.

Высокая доступность

GKE предоставляет высокодоступный кластер Kubernetes промышленного уровня, не требующий от вас затрат на настройку и обслуживание, как в случае с собственной локальной инфраструктурой. Всем можно управлять через Google Cloud API, используя Deployment Manager[1], Terraform или другие инструменты, а также веб-консоль GCP. Естественно, платформа GKE полностью интегрирована со всеми остальными сервисами Google Cloud.

Для дополнительных гарантий высокой доступности можно создавать *многозонные* кластеры, которые распределяют рабочие узлы по нескольким зонам отказа (что примерно эквивалентно отдельным вычислительным центрам). Ваши рабочие задания продолжат выполняться, даже если неполадки коснутся всей зоны отказа.

---

[1] Deployment Manager — утилита командной строки от Google, предназначенная для управления облачными ресурсами; не следует путать ее с Kubernetes Deployments.

Региональные кластеры идут еще дальше и распределяют между зонами отказа не только рабочие, но и ведущие узлы.

Автомасштабирование кластеров

GKE также предлагает привлекательную возможность автоматического масштабирования (см. пункт «Автомасштабирование» на с. 145). Если его включить для любых рабочих заданий, которые ждут появления доступных ресурсов, система автоматически будет добавлять новые узлы в соответствии с нагрузкой.

И наоборот, если появится лишняя пропускная возможность, система автомасштабирования сконцентрирует pod-оболочки на меньшем количестве узлов, избавившись от тех из них, которые не используются. Это помогает контролировать расходы, поскольку тарификация в GKE основана на количестве рабочих узлов.

Сервис GKE — лучший в своем роде

Google занимается Kubernetes дольше, чем кто бы то ни был, и это заметно. По нашему мнению, GKE является лучшим сервисом по управлению Kubernetes из всех доступных. Если у вас уже есть инфраструктура в Google Cloud, использовать GKE для работы с Kubernetes будет логично. Если же вы обосновались на другом облаке, это не должно вас удерживать от использования GKE — но лучше сперва ознакомиться с аналогичными сервисами от вашего облачного провайдера.

Но если вы все еще не определились с выбором облака, GKE может стать убедительным аргументом в пользу Google Cloud.

Amazon Elastic Container Service для Kubernetes

Amazon также давно предоставляет услуги по управлению кластерами контейнеров, однако до недавних пор единственным вариантом у нее был сервис Elastic Container Service (ECS) на основе собственных проприетарных технологий.

Система ECS (aws.amazon.com/ecs/) вполне подходит для использования, но она не настолько мощная и гибкая, как Kubernetes, и, по всей видимости, даже Amazon решила, что будущее за последней, судя по запуску Elastic Container Service для Kubernetes (EKS) (да, по логике вещей EKS должно расшифровываться как *Elastic Kubernetes Service*, но это не так).

Этот сервис не так и прост в использовании (blog.hasura.io/gke-vs-aks-vs-eks-411f080640dc), если сравнивать его с Google Kubernetes Engine, поэтому будьте готовы потратить больше усилий на подготовку. К тому же, в отличие от конкурентов, EKS взимает плату за ведущие узлы и другую инфраструктуру кластера, и это делает его более дорогим, чем сервисы по управлению Kubernetes от Google и Microsoft, с учетом кластеров одинакового размера.

Если у вас уже есть инфраструктура в AWS или контейнеризированные рабочие задания в более старом сервисе ECS, которые вы хотите перенести на Kubernetes, EKS будет разумным выбором. Но, так как это самое новое предложение на рынке Kubernetes, ему еще есть куда расти, чтобы сравняться с конкурентами от Google и Microsoft.

Azure Kubernetes Service (AKS)

Компания Microsoft пришла на облачный рынок немного позже, чем Amazon и Google, но стремительно наверстывает упущенное. Azure Kubernetes Service (AKS) (azure.microsoft.com/en-us/services/kubernetes-service) предлагает большинство тех возможностей, которыми обладают и конкуренты вроде GKE от Google. Вы можете создать кластер из веб-интерфейса или с помощью утилиты командной строки Azure.

По аналогии с GKE и EKS вы не имеете доступа к ведущим узлам, которые управляются изнутри, а тарификация основана на количестве рабочих узлов в вашем кластере.

OpenShift

OpenShift (www.openshift.com) — это не просто сервис по управлению Kubernetes, а полноценный продукт типа «платформа как услуга» (patform as a service, PaaS), нацеленный на управление всем жизненным циклом разработки, включающий системы непрерывной интеграции, развертывания приложений, мониторинга и оркестрации, а также инструменты сборки и программу запуска тестов.

OpenShift можно развертывать на физических серверах, виртуальных машинах, приватных и публичных облаках. Благодаря этому вы можете создать кластер Kubernetes, который охватывает все среды. Это делает данную платформу хорошим выбором для очень больших организаций или компаний с гетерогенной инфраструктурой.

IBM Cloud Kubernetes Service

Конечно же, почтенная компания IBM не могла остаться в стороне от области услуг по управлению Kubernetes. Сервис IBM Cloud Kubernetes Service (www.ibm.com/cloud/container-service) довольно прост и понятен. Он позволяет подготовить обычный кластер Kubernetes в IBM Cloud.

Помимо простого графического интерфейса, для доступа к кластеру IBM Cloud и управления им можно использовать сразу две утилиты командной строки: стандартную, входящую в состав Kubernetes, и ту, что предоставляется облаком. Сервис IBM не предлагает каких-либо потрясающих воображение возможностей, которые бы отличали его от основных конкурентов, но это логичный выбор для тех, кто уже использует IBM Cloud.

Heptio Kubernetes Subscription (HKS)

Heptio Kubernetes Subscription (HKS) (heptio.com/products/kubernetes-subscription) рассчитан на крупные предприятия, которым нужна безопасность и гибкость работы кластеров сразу в нескольких публичных облаках.

Компания Heptio имеет солидную репутацию в мире Kubernetes: ее руководители Крэйг Маклаки и Джо Беда являются соавторами этого сервиса, на их счету множество важных инструментов с открытым исходным кодом, таких как Velero (см. подраздел «Velero» на с. 268) и Sonobuoy (см. подраздел «Проверка на соответствие с помощью Sonobuoy» на с. 148).

Решения для Kubernetes под ключ

Сервисы по управлению Kubernetes вполне отвечают большинству бизнес-требований, но бывают определенные обстоятельства, в которых такие сервисы даже не рассматриваются. Растет количество предложений *под ключ*, когда нажатием кнопки в браузере можно получить готовый к использованию кластер Kubernetes промышленного уровня.

Такие предложения являются привлекательными как для крупных предприятий, которые могут поддерживать коммерческие отношения с поставщиком, так и для небольших компаний с маленьким штатом инженеров и системных администраторов. Ниже перечислено несколько вариантов из этой ниши.

Stackpoint

Stackpoint (stackpoint.io) рекламируется как «самый простой способ развернуть кластер Kubernetes в публичном облаке»: всего за три щелчка кнопкой мыши. Вам доступны разные тарифные планы, начиная с $50 в месяц. При этом Stackpoint предлагает кластеры с неограниченным числом узлов, высокой доступностью для ведущих узлов и `etcd`, а также поддержку *федеративных* кластеров, которые могут охватывать разные облака (см. пункт «Федеративные кластеры» на с. 140).

Stackpoint является компромиссным решением между самостоятельным размещением Kubernetes и полностью управляемыми сервисами. Это привлекательное решение для компаний, которые хотят иметь возможность запускать и обслуживать Kubernetes из веб-портала, но так, чтобы рабочие узлы находились в их собственной облачной инфраструктуре.

Containership Kubernetes Engine (CKE)

CKE (blog.containership.io/introducing-containership-kubernetes-engine) — еще один веб-интерфейс для запуска Kubernetes в публичном облаке. Он позволяет получить готовый кластер с разумными параметрами по умолчанию, почти любой из которых можно изменить в соответствии с более жесткими требованиями.

Установщики Kubernetes

Если вам не подходят готовые или управляемые кластеры, стоит подумать о самостоятельном размещении Kubernetes, то есть о подготовке и запуске Kubernetes своими силами на собственных компьютерах.

Вряд ли вы будете развертывать и запускать Kubernetes полностью с нуля, разве что в обучающих или демонстрационных целях. Большинство людей используют один или несколько из доступных установщиков и сервисов для управления своими кластерами.

kops

kops (kubernetes.io/docs/setup/production-environment/tools/kops) — это утилита командной строки для автоматического выделения кластеров. Она входит в состав проекта

Kubernetes и уже давно служит инструментом, предназначенным специально для AWS. В настоящий момент начинается бета-тестирование поддержки Google Cloud, также планируется совместимость с другими провайдерами.

Утилита kops позволяет создавать высокодоступные кластеры и поэтому подходит для развертывания Kubernetes в промышленных условиях. Она использует декларативную конфигурацию, такую же, как и ресурсы Kubernetes. Помимо выделения необходимых облачных ресурсов и подготовки кластера, она также способна выполнять масштабирование вверх и вниз, изменять размеры узлов, выполнять обновления и решать другие полезные административные задачи.

Как и все остальное в мире Kubernetes, kops находится на стадии активной разработки. Тем не менее это проверенный и многофункциональный инструмент, который широко используется. Если вы планируете самостоятельно запускать Kubernetes в AWS, kops будет хорошим выбором.

Kubespray

Проект Kubespray (github.com/kubernetes-sigs/kubespray), ранее известный как Kargo, находится под покровительством Kubernetes. Это инструмент для простого развертывания промышленных кластеров. Он предлагает множество возможностей, включая высокую доступность и поддержку нескольких платформ.

Kubespray фокусируется на установке Kubernetes на существующие компьютеры, особенно на локальные и физические серверы. Но также этот инструмент подходит для любой облачной среды, включая приватные облака (виртуальные машины, запущенные на ваших собственных серверах).

TK8

TK8 (github.com/kubernauts/tk8) — утилита командной строки для создания кластеров Kubernetes, которая использует как Terraform (для создания облачных серверов), так и Kubespray (для установки на них Kubernetes). Она написана на Go (естественно) и поддерживает установку на AWS, OpenStack и «чистые серверы» (bare-metal servers); в планах добавление совместимости с Azure и Google Cloud.

Помимо построения кластеров Kubernetes, TK8 устанавливает дополнения, такие как Jmeter Cluster для нагрузочного тестирования, Prometheus для мониторинга, Jaeger, Linkerd или Zipkin для трассирования, Ambassador API Gateway с Envoy для

условного контроля доступа и балансирования нагрузки, Istio для межсервисного взаимодействия, Jenkins-X for CI/CD и Helm или Kedge для управления пакетами в Kubernetes.

Kubernetes: трудный путь

Учебник *Kubernetes The Hard Way* (github.com/kelseyhightower/kubernetes-the-hard-way) от Келси Хайтауэр — это скорее не руководство по подготовке и установке Kubernetes, а субъективный путеводитель по процессу построения кластеров, который иллюстрирует всю сложность разных его аспектов. Тем не менее он очень поучителен и его стоит изучить любому, кто собирается использовать Kubernetes — даже если в виде управляемого сервиса, просто чтобы получить представление о том, как там внутри все работает.

kubeadm

Утилита kubeadm (kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm) входит в состав дистрибутива Kubernetes и должна помочь вам устанавливать и обслуживать кластеры в соответствии с лучшими рекомендациями. kubeadm не занимается выделением инфраструктуры для самого кластера, поэтому она подходит для установки Kubernetes на физические и облачные серверы любого рода.

Многие инструменты и сервисы, о которых мы еще поговорим в этой главе, используют kubeadm для выполнения административных операций, но если вам хочется, ничто не мешает вам применять эту утилиту напрямую.

Tarmak

Tarmak (blog.jetstack.io/blog/introducing-tarmak) — это инструмент для управления жизненным циклом кластеров Kubernetes, который должен упростить и сделать более надежными модификацию и обновления кластерных узлов. Многие инструменты просто заменяют узлы, что нередко занимает много времени и требует перемещения больших объемов данных между серверами в процессе пересборки: Tarmak же восстанавливает или обновляет уже имеющийся узел.

Tarmak использует Terraform для выделения узлов кластера и Puppet для управления конфигурацией на самих узлах. Благодаря этому внесение изменений в конфигурацию узлов происходит быстрее и безопаснее.

Rancher Kubernetes Engine (RKE)

RKE (github.com/rancher/rke) стремится быть простым и быстрым установщиком Kubernetes. Этот инструмент не занимается выделением узлов, поэтому перед его использованием вы должны самостоятельно установить Docker на свои узлы. RKE поддерживает высокую доступность управляющего уровня Kubernetes.

Модуль Puppet для Kubernetes

Puppet — это мощный, зрелый и многофункциональный инструмент управления общего назначения, который имеет очень широкое распространение и большую экосистему модулей с открытым исходным кодом. Официально поддерживаемый модуль (forge.puppet.com/puppetlabs/kubernetes) устанавливает и конфигурирует Kubernetes на существующих узлах, предлагая поддержку высокой доступности как для управляющего уровня, так и для `etcd`.

Kubeformation

Kubeformation (kubeformation.sh) — это онлайн-конфигуратор для Kubernetes. Вы выбираете параметры для своего кластера с помощью веб-интерфейса, и он генерирует конфигурационные шаблоны для API автоматизации вашего конкретного облачного провайдера (например, Deployment Manager для Google Cloud или Azure Resource Manager for Azure). В планах есть поддержка других облачных хранилищ.

Возможно, Kubeformation в использовании не самый простой инструмент, но, поскольку он является оболочкой существующих средств автоматизации, таких как Deployment Manager, ему присуща повышенная гибкость. Например, если вы уже управляете своей инфраструктурой в Google Cloud с помощью Deployment Manager, Kubeformation идеально впишется в ваш рабочий процесс.

Покупать или строить: наши рекомендации

Мы намеренно сделали этот обзор доступных решений для управления кластерами Kubernetes кратким, поскольку диапазон предложений очень большой и разнообразный и постоянно расширяется. Тем не менее можем дать несколько советов, основанных на здравом смысле. Один из них опирается на философию использования как можно меньшего количества ПО (www.intercom.com/blog/run-less-software).

Не используйте слишком много ПО

> Философия использования как можно меньшего количества ПО стоит на трех китах, каждый из которых поможет вам управлять временем и побеждать врагов.
>
> 1. Выберите стандартную технологию.
> 2. Делегируйте рутинные, однообразные задачи.
> 3. Создайте устойчивое конкурентное преимущество.
>
> *Рич Арчболд*

Хотя использование инновационных технологий достаточно веселое и увлекательное занятие, но с точки зрения бизнеса не всегда имеющее смысл. Хорошим выбором будет применение *скучного* ПО, которым пользуются все остальные. Оно, скорее всего, работает и хорошо поддерживается, и вам не придется брать на себя риски и хлопоты, связанные с неизбежными ошибками в коде.

Если вы выполняете контейнеризированные рабочие задания и создаете облачно-ориентированные приложения, Kubernetes будет скучным вариантом в самом лучшем смысле этого слова. Исходя из этого, вам следует выбирать наиболее зрелые, стабильные и широко распространенные инструменты и сервисы для Kubernetes.

Рутинные, однообразные задачи (undifferentiated heavy lifting) — это термин, предложенный компанией Amazon для обозначения любой сложной работы, такой как установка и управление ПО, обслуживание инфраструктуры и т. д. В этой работе нет ничего особенного, она одинаковая и для вас, и для любой другой компании. Она стоит денег и не генерирует доход.

Согласно философии *использования как можно меньшего количества ПО* вы должны делегировать рутинные, однообразные задачи за пределы компании; это будет дешевле в долгосрочной перспективе и освободит ресурсы, которые можно будет направить на ваш основной бизнес.

По возможности используйте Kubernetes в виде управляемого сервиса

Учитывая принципы *использования как можно меньшего количества ПО*, мы советуем вам вынести администрирование кластера Kubernetes в управляемый сервис. Установка, конфигурация, обслуживание, защита, обновление и обеспечение на-

дежности *вашего* кластера — это рутинные, однообразные задачи, поэтому почти любой компании имеет смысл отказаться от их самостоятельного выполнения.

На рынке управляемых решений для Kubernetes сервис Google Kubernetes Engine (GKE) является очевидным фаворитом. Возможно, другие облачные провайдеры подтянутся через год или два, но Google все еще далеко впереди и будет удерживать лидерство в ближайшем будущем.

Если вам нельзя зависеть от одного облачного провайдера и вы хотите получать круглосуточную техническую поддержку от именитой компании, стоит обратить внимание на Heptio Kubernetes Subscription.

Если в качестве одной из услуг вам нужна высокая доступность для управляющего уровня вашего кластера, но в то же время вы хотите иметь гибкость в запуске собственных рабочих узлов, подумайте об использовании Stackpoint.

Но что насчет привязки к поставщику?

Если вы остановитесь на сервисе по управлению Kubernetes от какого-то определенного поставщика (например, Google Cloud), привяжет ли это вас к определенному облаку и ограничит ли ваш выбор в будущем? Не обязательно. Kubernetes является стандартной платформой, поэтому любые приложения и сервисы, которые вы создадите с расчетом на Google Kubernetes Engine, будут работать и на других системах от сертифицированных поставщиков Kubernetes. Выбор Kubernetes — уже большой шаг к тому, чтобы избежать привязки к поставщику.

Подталкивают ли вас управляемые сервисы к такой привязке, если сравнивать с самостоятельным размещением Kubernetes? Мы считаем, что наоборот. Локальное размещение Kubernetes требует обслуживания большого объема оборудования и конфигурации, тесно связанных с API определенного облачного провайдера. Например, выделение виртуальных машин для запуска Kubernetes требует совершенно разного кода в AWS и Google Cloud. Некоторые установщики из числа тех, что мы упоминали в этой главе, поддерживают нескольких облачных провайдеров, но многие такой поддержки не имеют.

Одной из задач Kubernetes является абстрагирование от технических подробностей облачной платформы и предоставление разработчикам стандартного, знакомого интерфейса, который работает одинаково и на Azure, и на Google Cloud. Если вы проектируете свои приложения и автоматику с расчетом на Kubernetes, а не на внутреннюю облачную инфраструктуру, вы свободны от привязки к поставщику (в разумных пределах).

При необходимости используйте стандартные инструменты Kubernetes для самостоятельного размещения

Самостоятельное размещение Kubernetes следует рассматривать только в случае, если к вам предъявляются специальные требования, исключающие использование управляемых решений.

В таких ситуациях стоит выбирать наиболее проверенные, мощные и широко используемые инструменты. Мы рекомендуем kops или Kubespray, в зависимости от ваших потребностей.

Используйте kops, если уверены, что в долгосрочной перспективе будете работать только с одним поставщиком (в особенности если это AWS).

Но если ваш кластер должен охватывать несколько облачных хранилищ или платформ, включая физические серверы, и вы хотите оставить себе свободу выбора, советуем выбрать Kubespray.

Если ваш выбор ограничен

Причины, по которым вам не подходят сервисы по управлению Kubernetes, могут иметь отношение скорее к бизнесу, а не к техническим деталям. Если вы являетесь партнером хостинговой компании или облачного провайдера, у которых нет соответствующих сервисов, очевидно, это ограничит ваш выбор.

Однако вместо этого вам, возможно, подойдут готовые решения под ключ, такие как Stackpoint или Containership. Они берут на себя обслуживание ваших ведущих узлов, но подключают их к рабочим узлам, которые запущены в рамках вашей собственной инфраструктуры. Это хороший компромисс, учитывая, что большая часть накладных расходов, связанных с системным администрированием Kubernetes, относится к обслуживанию ведущих узлов.

Локально размещенные физические серверы

Возможно, вы удивитесь, но облачная ориентированность не обязательно подразумевает размещение *в облаке*, то есть делегирование вашей инфраструктуры публичному облачному провайдеру, такому как Azure или AWS.

Многие организации полностью или частично размещают свою инфраструктуру на физическом оборудовании — либо совместно в вычислительных центрах, либо

локально. Все, что было сказано в этой книге о Kubernetes и контейнерах, относится в равной степени и к облаку, и к внутренней инфраструктуре.

Вы можете запускать Kubernetes на своих собственных физических серверах — если ваш бюджет ограничен, можете использовать для этого даже стопку Raspberry Pi (рис. 3.3). Некоторые компании имеют *приватные облака*, которые состоят из виртуальных машин, размещенных на локальных компьютерах.

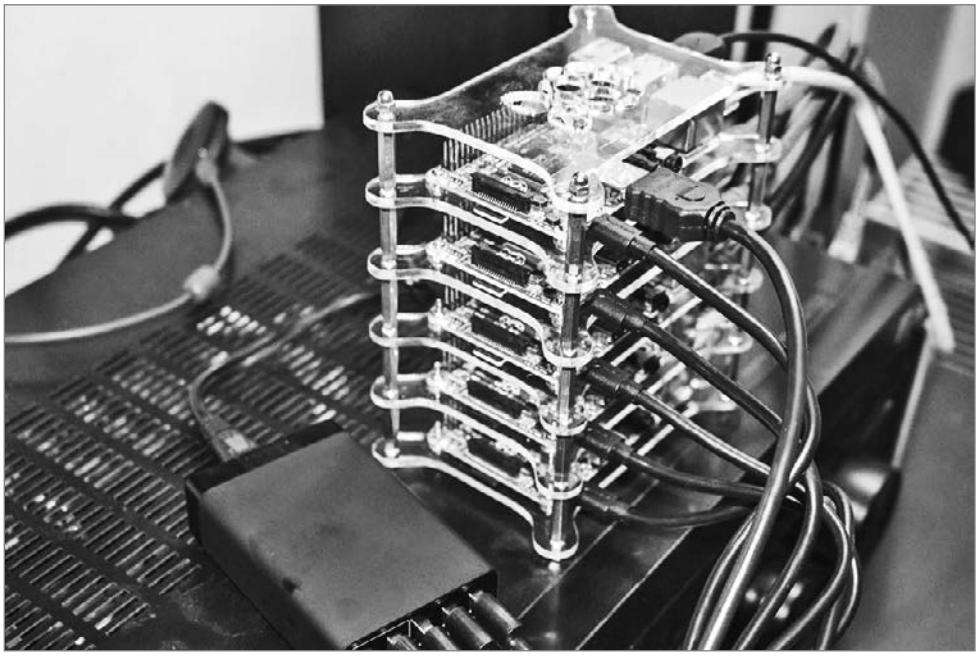

Рис. 3.3. Бюджетный Kubernetes: кластер из Raspberry Pi (фото Дэвида Меррика)

Бескластерные контейнерные сервисы

Если вы действительно хотите минимизировать накладные расходы на выполнение рабочих заданий в контейнерах, можете не останавливаться на полностью управляемых сервисах, а идите еще дальше. Речь о так называемых *бескластерных* сервисах, таких как Azure Container Instances или Fargate от Amazon. И хотя внутри все же есть какой-то кластер, у вас нет к нему доступа через такие инструменты, как, например, `kubectl`. Вместо этого вы просто указываете образ контейнера, который нужно выполнить, и несколько параметров (например, требования вашего приложения к процессору и памяти), а сервис делает все остальное.

Amazon Fargate

Цитируя Amazon, «Fargate — это как EC2, только вместо виртуальной машины вы получаете контейнер». В отличие от ситуации с ECS вам не нужно самостоятельно выделять узлы кластера и затем подключать их к управляющему уровню. Вы просто определяете задание, которое представляет собой набор инструкций о том, как запускать ваш образ контейнера, и запускаете его на выполнение. Для вашего задания действует посекундная тарификация процессорного времени и памяти.

Наверное, будет справедливо сказать, что Fargate (aws.amazon.com/blogs/aws/aws-fargate) имеет смысл для простых автономных долгоиграющих вычислительных или пакетных задач (таких как обработка данных), не требующих особой настройки или интеграции с другими сервисами. Этот сервис также идеально подходит для контейнеров-сборщиков, которые обычно быстро завершают работу, и для любой другой ситуации, когда накладные расходы на управление рабочими узлами не являются оправданными.

Если вы уже используете ECS с рабочими узлами на EC2, переход на Fargate освободит вас от необходимости выделения и администрирования этих узлов. На сегодня сервис Fargate доступен в некоторых регионах для выполнения ECS-заданий; поддержка EKS появилась в декабре 2019 года.

Azure Container Instances (ACI)

Сервис Azure Container Instances (ACI) (azure.microsoft.com/en-gb/services/container-instances) от Microsoft похож на Fargate, но при этом предлагает интеграцию с Azure Kubernetes Service (AKS). Например, вы можете сконфигурировать кластер AKS для временного выделения дополнительных pod-оболочек внутри ACI, чтобы справиться со скачками нагрузки.

Точно так же вы можете при необходимости запускать в ACI пакетные задания; при этом узлы не будут простаивать, если для них нет работы. Microsoft называет это *бессерверными контейнерами*, но нам кажется, что такой термин и сбивает с толку (слово *«бессерверный»* обычно указывает на облачные функции или FaaS), и является неточным (серверы присутствуют, просто у вас нет к ним доступа).

Сервис ACI также интегрирован с Azure Event Grid — сервисом по управлению маршрутизацией событий от Microsoft. Благодаря Event Grid контейнеры ACI могут взаимодействовать с облачными сервисами, облачными функциями или приложениями Kubernetes, запущенными в AKS.

С помощью Azure Functions можно создавать и запускать контейнеры ACI и передавать им данные. Преимущество этого подхода в том, что вы можете выполнить любое рабочее задание из облачной функции, не ограничиваясь официально поддерживаемыми (благословленными) языками, такими как Python или JavaScript.

Если ваше рабочее задание можно поместить в контейнер, оно может выполняться в виде облачной функции со всем сопутствующим инструментарием. Например, Microsoft Flow позволяет выстраивать рабочие задания графическим образом, соединяя контейнеры, функции и события, и это могут делать даже люди, далекие от программирования.

Резюме

Куда ни глянь, везде Kubernetes! Наше путешествие по обширному списку инструментов, сервисов и продуктов для Kubernetes было вынужденно коротким, но, надеемся, полезным для вас.

Мы пытались сделать обзор продуктов и возможностей как можно более актуальным, но мир очень быстро меняется и вполне можно ожидать, что многое изменится даже к моменту, когда вы будете читать эти строки.

Тем не менее нам кажется, что основная идея останется неизменной: самостоятельное управление кластером Kubernetes не имеет смысла, поставщик сервисов может сделать это лучше и дешевле.

Наш опыт консалтинга компаний, переходящих на Kubernetes, показывает, что подобный подход многим кажется неожиданным: по крайней мере, мысль о таком приходит в голову далеко не всем. Мы регулярно сталкиваемся с организациями, которые уже предприняли первые шаги на пути к самостоятельному размещению кластеров с помощью таких инструментов, как kops, но при этом даже не задумывались об использовании управляемых сервисов вроде GKE. И очень даже зря.

Вам также надлежит помнить о следующем.

- Кластеры Kubernetes состоят из *ведущих узлов*, на которых выполняется *управляющий уровень*, и *рабочих узлов*, на которых выполняются рабочие задания.
- Промышленные кластеры должны быть *высокодоступными*: отказ ведущего узла не должен приводить к потере данных или влиять на работу кластера.
- Существует большая разница между простым демонстрационным кластером и кластером, готовым к критически важным промышленным нагрузкам. Нужно обеспечить высокую доступность, безопасность и управление узлами.

- Управление собственными кластерами требует много времени, усилий и опыта. Но даже это не застрахует вас от ошибок.
- Управляемые сервисы, такие как Google Kubernetes Engine, делают всю тяжелую работу за вас и требуют гораздо меньше затрат, чем вариант с самостоятельным размещением.
- Готовые решения под ключ являются хорошим компромиссом между самостоятельным размещением и полностью управляемой конфигурацией Kubernetes. Такие поставщики, как Stackpoint, берут на себя управление вашими ведущими узлами, позволяя вам запускать рабочие узлы на собственных компьютерах.
- Если вам приходится самостоятельно размещать собственный кластер, можно использовать kops: это проверенный и широко применяемый инструмент, позволяющий создавать и администрировать кластеры промышленного уровня на AWS и Google Cloud.
- По возможности следует использовать управляемые сервисы Kubernetes. Для большинства компаний это лучший выбор с точки зрения стоимости, накладных расходов и качества.
- Если вам не подходят управляемые сервисы, готовые решения под ключ могут стать хорошим компромиссом.
- Не размещайте свой кластер самостоятельно без веских причин, связанных с бизнесом. В случае самостоятельного размещения не стоит недооценивать инженерные ресурсы, которые потребуются для начальной настройки и последующего обслуживания.

4 Работа с объектами Kubernetes

> Не могу понять, почему людей пугают новые идеи.
> Лично меня пугают старые.
>
> *Джон Кейдж*

В главе 2 вы создали и развернули приложение в Kubernetes. В данной главе вы изучите фундаментальные объекты Kubernetes, которые участвуют в этом процессе: pod-оболочки, развертывания и сервисы. Вы также научитесь использовать незаменимый инструмент Helm для управления приложениями в Kubernetes.

После выполнения примера из подраздела «Запуск демонстрационного приложения» на с. 61 у вас должен быть образ контейнера, который выполняется в кластере Kubernetes. Но как это все на самом деле работает? Неявно для пользователя команда `kubectl run` создает ресурс Kubernetes под названием *«развертывание»* (deployment). Что он собой представляет? И как на самом деле развертывание запускает образ контейнера?

Развертывания

Вспомните, как мы выполняли демонстрационное приложение с помощью Docker. Команда `docker container run` запустила контейнер, который работал до тех пор, пока вы его не уничтожили, набрав `docker stop`.

Но представьте, что контейнер завершает работу по какой-то другой причине: возможно, случился сбой в программе, произошла системная ошибка, на вашем компьютере закончилось дисковое пространство или же космический луч попал в ваш центральный процессор в неудачный момент (маловероятно, но бывает).

Предположим, что приложение промышленное: это означает, что, пока кто-то не зайдет в терминал и не перезапустит контейнер с помощью команды `docker container run`, у вас будут недовольные пользователи.

Это не очень удачный механизм. На самом деле вам нужна какая-то наблюдающая программа, которая станет постоянно проверять, работает ли контейнер, а если тот остановится, немедленно запустит его. На традиционных серверах для этого можно использовать такие инструменты: `systemd`, `runit` или `supervisord`. Docker предлагает нечто подобное, и у Kubernetes, что неудивительно, тоже есть функция-супервизор — *развертывание*.

Надзор и планирование

Для каждой программы, за которой нужно следить, Kubernetes создает соответствующий объект Deployment, записывающий некоторую связанную с ней информацию: имя образа контейнера, количество реплик (копий), которые вы хотите выполнять, и любые другие параметры, необходимые для запуска контейнера.

В связке с ресурсом Deployment работает некий объект Kubernetes под названием *контроллер*. Контроллеры отслеживают ресурсы, за которые отвечают, убеждаясь в том, что те присутствуют и выполняются, а если заданное развертывание по какой-либо причине не имеет достаточного количества реплик, дополнительно их создают. Если же реплик слишком много, контроллер уберет лишние — так или иначе, он следит за тем, чтобы реальное состояние совпадало с желаемым.

На самом деле развертывание не управляет репликами напрямую: вместо этого оно автоматически создает сопутствующий объект под названием `ReplicaSet`, который сам этим занимается. Мы поговорим об объектах `ReplicaSet` чуть позже, в соответствующем разделе на с. 93, но сначала поближе познакомимся с развертываниями — в основном с ними вы и будете иметь дело.

Перезапуск контейнеров

На первый взгляд поведение развертывания может показаться неожиданным. Если ваш контейнер завершит работу и остановится, развертывание его перезапустит. То же самое произойдет в случае сбоя или если вы уничтожите контейнер по сигналу или с помощью `kubectl` (так в теории, хотя в реальности все немного сложнее, и скоро вы в этом убедитесь).

Большинство приложений Kubernetes должны работать долго и надежно, поэтому подобное поведение имеет смысл: контейнер может остановиться по разным при-

чинам, и в большинстве случаев реакцией живого оператора будет перезапуск — именно так по умолчанию и ведет себя Kubernetes.

Это поведение можно менять для отдельных контейнеров: например, их можно никогда не перезапускать или разрешить перезапуск только в случае сбоя, а не нормального завершения работы (см. раздел «Политики перезапуска» на с. 202). Однако поведение по умолчанию (всегда перезапускать) обычно подходит.

Задача развертывания состоит в том, чтобы отслеживать связанные с ним контейнеры и постоянно поддерживать определенное их количество. Если контейнеров меньше, чем указано, оно запустит еще. Если их слишком много, оно уничтожит некоторые из них. Это намного более мощный и гибкий механизм, чем традиционные программы-супервизоры.

Обращение к развертываниям

Чтобы просмотреть все активные развертывания в вашем текущем пространстве имен (см. раздел «Использование пространств имен» на с. 117), введите следующую команду:

```
kubectl get deployments
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
demo      1         1         1            1           21h
```

Для более подробной информации об этом конкретном развертывании используйте такую команду:

```
kubectl describe deployments/demo
Name:                   demo
Namespace:              default
CreationTimestamp:      Tue, 08 May 2018 12:20:50 +0100
...
```

Как видите, мы получили много информации, большая часть которой нам пока не нужна. Давайте более пристально рассмотрим раздел `Pod Template`:

```
Pod Template:
  Labels:  app=demo
  Containers:
   demo:
    Image:        cloudnatived/demo:hello
    Port:         8888/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
```

Как вы знаете, развертывание содержит информацию, которая нужна Kubernetes для выполнения контейнера, и она здесь представлена. Но что такое `Pod Template`? Прежде чем ответить на этот вопрос, давайте разберемся, что такое `Pod`.

Pod-оболочки

`Pod` — это объект Kubernetes, который представляет группу из одного или нескольких контейнеров (в английском языке термин *pod* обозначает группу китов, и это соответствует околомореходной окраске метафор Kubernetes). Мы будем называть его *pod-оболочкой*.

Почему бы развертыванию не управлять отдельными контейнерами напрямую? Дело в том, что иногда контейнеры должны запускаться вместе, работать на одном узле, взаимодействовать локально и, возможно, использовать одно хранилище данных.

Например, у приложения для ведения блога может быть один контейнер для синхронизации содержимого с Git-репозиторием и еще один с веб-сервером Nginx, который раздает содержимое блога пользователям. Поскольку они обмениваются данными, выполнение этих двух контейнеров должно быть запланировано в одной pod-оболочке. Хотя на практике многие pod-оболочки содержат лишь один контейнер, как в нашем случае (подробнее об этом в подразделе «Что должно находиться внутри pod-оболочки» на с. 186).

Итак, спецификация pod-оболочки содержит список контейнеров. В нашем примере есть лишь один контейнер, `demo`:

```
demo:
 Image:        cloudnatived/demo:hello
 Port:         8888/TCP
 Host Port:    0/TCP
 Environment:  <none>
 Mounts:       <none>
```

В вашем случае параметр `Image` будет равен **`YOUR_DOCKER_ID`**`/myhello`, и вместе с номером порта этой информации достаточно, чтобы развертывание могло запустить pod-оболочку и поддерживать ее работу.

И это важный момент: команда `kubectl run` создает не pod-оболочку напрямую, а развертывание, которое затем запускает `Pod`-объект. Развертывание — это декларация того состояния, которое вам нужно: «pod-оболочка должна быть запущена с контейнером `myhello` внутри».

Объекты ReplicaSet

Мы упомянули, что развертывания запускают pod-оболочки, но в реальности все немного сложнее. На самом деле развертывания не управляют pod-оболочками напрямую, это обязанность объекта `ReplicaSet`.

`ReplicaSet` отвечает за группу идентичных pod-оболочек (или *реплик*). Если обнаружится, что pod-оболочек слишком мало (или много) по сравнению со спецификацией, контроллер `ReplicaSet` запустит новые (или остановит существующие) копии, чтобы исправить ситуацию.

Развертывания, в свою очередь, управляют объектами `ReplicaSet` и контролируют поведение реплик в момент их обновления — например, при выкатывании новой версии вашего приложения (см. раздел «Стратегии развертывания» на с. 302). Когда вы обновляете развертывание, для управления pod-оболочками создается новый объект `ReplicaSet`; по завершении обновления старая копия `ReplicaSet` уничтожается вместе со своими pod-оболочками.

На рис. 4.1 каждый объект `ReplicaSet` (V1, V2, V3) представляет отдельную версию приложения с соответствующими pod-оболочками.

Обычно с `ReplicaSet` не нужно взаимодействовать напрямую, так как развертывания делают это за вас. Но все же не помешает знать, что это такое.

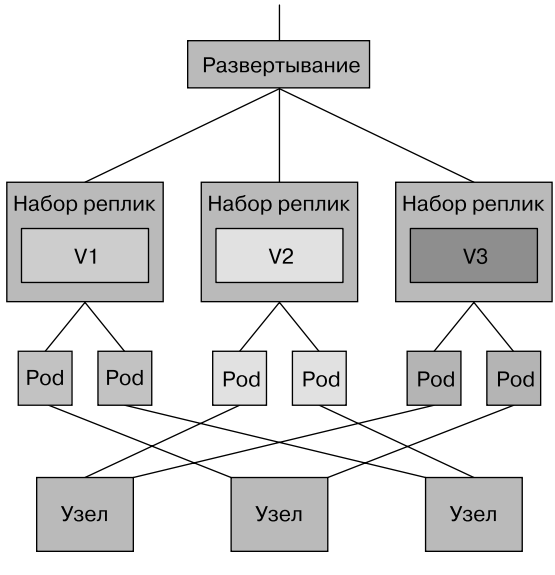

Рис. 4.1. Развертывание, экземпляры ReplicaSet и pod-оболочки

Поддержание желаемого состояния

Контроллеры Kubernetes непрерывно сравнивают желаемое состояние, указанное каждым ресурсом, с реальным состоянием кластера и вносят необходимые корректировки. Этот процесс называют *циклом согласования*, поскольку он все время повторяется в попытке согласовать текущее состояние с желаемым.

Например, при первом создании развертывания `demo` Kubernetes немедленно запускает одноименную pod-оболочку. Если она собирается останавливаться, а развертывание все еще существует, Kubernetes запустит ее снова.

Проверим это прямо сейчас, остановив pod-оболочку вручную. Для начала убедитесь в том, что она в самом деле запущена:

```
kubectl get pods --selector app=demo
NAME                    READY     STATUS    RESTARTS   AGE
demo-54df94b7b7-qgtc6   1/1       Running   1          22h
```

Теперь остановите `Pod`-объект с помощью следующей команды:

```
kubectl delete pods --selector app=demo
pod "demo-54df94b7b7-qgtc6" deleted
```

Снова выведите список pod-оболочек:

```
kubectl get pods --selector app=demo
NAME                    READY     STATUS        RESTARTS   AGE
demo-54df94b7b7-hrspp   1/1       Running       0          5s
demo-54df94b7b7-qgtc6   0/1       Terminating   1          22h
```

Вы можете видеть, что оригинальная pod-оболочка останавливается (со статусом `Terminating`), но ее место уже занимает новая, которой всего пять секунд. Так работает цикл согласования.

Создав развертывание, вы сообщили Kubernetes о том, что pod-оболочка `demo` должна работать *всегда*. Система ловит вас на слове, и, даже если вы сами удалите этот `Pod`-объект, она посчитает, что вы наверняка ошиблись, и услужливо запустит замену.

Закончив экспериментировать с развертыванием, остановите и уничтожьте его с помощью такой команды:

```
kubectl delete all --selector app=demo
pod "demo-54df94b7b7-hrspp" deleted
service "demo" deleted
deployment.apps "demo" deleted
```

Планировщик Kubernetes

Ранее мы сказали, что *развертывание создаст pod-оболочки, а Kubernetes при необходимости их запустит*, но не объяснили, как такое происходит.

За эту часть процесса отвечает компонент Kubernetes под названием *«планировщик»*. Когда развертывание (через соответствующий объект `ReplicaSet`) решит, что нужна новая реплика, оно создаст ресурс `Pod` в базе данных Kubernetes. Одновременно с этим указанный ресурс добавляется в очередь — этакий ящик входящих сообщений для планировщика.

Задача планировщика — следить за этой очередью, взять из нее следующую запланированную pod-оболочку и найти узел, на котором ее можно запустить. При выборе подходящего узла (при условии, что такой имеется) планировщик будет исходить из нескольких разных критериев, включая ресурсы, запрашиваемые pod-оболочкой (подробнее об этом процессе мы поговорим в главе 5).

Как только выполнение pod-оболочки было запланировано, утилита `kubelet`, работающая на соответствующем узле, подхватывает ее и производит запуск ее контейнера (см. подраздел «Компоненты узла» на с. 66).

Когда вы удалили pod-оболочку в разделе «Поддержание желаемого состояния» на с. 94, утилита `kubelet` на соответствующем узле это заметила и инициировала замену. Она *знает*, что на этом узле должна работать pod-оболочка `demo`, а если таковая не обнаружена, ее следует запустить. А что бы случилось, если б вы выключили весь узел целиком? Его pod-оболочки вернулись бы обратно в очередь планировщика и были бы назначены другим узлам.

Инженер из компании Stripe Джулия Эванс написала восхитительно понятное объяснение того, как работает планировщик в Kubernetes (jvns.ca/blog/2017/07/27/how-does-the-kubernetes-scheduler-work).

Манифесты ресурсов в формате YAML

Итак, теперь вы знаете, как запускать приложения в Kubernetes. На этом все, мы закончили? Не совсем. Использование команды `kubectl run` для создания развертываний хоть и полезно, но имеет ограничения. Представьте, что вам хочется что-то поменять в спецификации развертывания: скажем, имя или версию образа. Вы могли бы удалить существующий объект Deployment (с помощью `kubectl delete`) и создать новый с подходящими полями. Но давайте посмотрим, не найдется ли лучшего способа.

Поскольку система Kubernetes по своей природе является *декларативной* и непрерывно согласовывает реальное состояние с желаемым, вам достаточно лишь поменять последнее (спецификацию развертывания), и Kubernetes все сделает за вас. Как это достигается?

Ресурсы являются данными

Все ресурсы Kubernetes, такие как развертывания и pod-оболочки, представлены записями во внутренней базе данных. Цикл согласования следит за любыми изменениями в записях и предпринимает соответствующие действия. На самом деле команда `kubectl run` лишь добавляет в базу данных новую запись о развертывании, а система делает все остальное.

Но для взаимодействия с Kubernetes вовсе не обязательно использовать команду `kubectl run` — вы можете создавать *манифесты* ресурсов (спецификацию их желаемого состояния) напрямую. Если хранить файл манифеста в системе контроля версий, то вместо выполнения императивных команд можно просто подменить этот файл и затем заставить Kubernetes прочитать обновленные данные.

Манифесты развертываний

Стандартным форматом для файлов манифестов в Kubernetes является YAML, хотя также поддерживается JSON. Как выглядит YAML-манифест развертывания?

В качестве примера возьмем наше демонстрационное приложение (`hello-k8s/k8s/deployment.yaml`):

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: demo
  labels:
    app: demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo
template:
  metadata:
    labels:
      app: demo
spec:
  containers:
    - name: demo
```

```
    image: cloudnatived/demo:hello
    ports:
    - containerPort: 8888
```

На первый взгляд выглядит сложно, но это в основном стереотипный код. Интересные участки содержат только ту информацию, которую вы уже видели в разных формах: имя и порт образа. Когда ранее вы передали данные параметры команде `kubectl run`, она неявно создала эквивалент этого YAML-манифеста и передала его Kubernetes.

Использование команды kubectl apply

Чтобы использовать всю мощь Kubernetes как декларативной системы типа *«инфраструктура как код»*, вы можете самостоятельно передавать YAML-манифесты вашему кластеру, используя команду `kubectl apply`.

Попробуйте это сделать на примере манифеста развертывания `hello-k8s/k8s/deployment.yaml`[1].

Выполните следующие команды в своей копии демонстрационного репозитория:

```
cd hello-k8s
kubectl apply -f k8s/deployment.yaml
deployment.apps "demo" created
```

Спустя несколько секунд должна заработать pod-оболочка `demo`:

```
kubectl get pods --selector app=demo
NAME                    READY     STATUS       RESTARTS   AGE
demo-6d99bf474d-z9zv6   1/1       Running      0          2m
```

Но мы еще не закончили, поскольку для подключения к pod-оболочке с помощью веб-браузера надо создать *сервис* — ресурс Kubernetes, который позволяет подключаться к развернутым pod-оболочкам (подробнее об этом — чуть позже).

Для начала посмотрим, что такое сервис и зачем он нам нужен.

Ресурсы типа «сервис»

Представьте, что вы хотите выполнить сетевое подключение к pod-оболочке (такой как наше демонстрационное приложение). Как это сделать? Вы могли бы узнать ее IP-адрес и подключиться напрямую по номеру порта. Но IP-адрес может

[1] k8s (произносится как «кейтс») — распространенное в мире Kubernetes сокращение, выполненное по гиковскому принципу нумеронима: первая и последняя буквы плюс количество символов между ними (k-8-s). См. также i18n (internationalization), a11y (accessibility) и o11y (observability).

измениться при перезапуске, поэтому вам придется запрашивать его снова и снова, чтобы увериться в том, что он актуален.

Хуже того, у pod-оболочки может быть несколько реплик и каждая со своим адресом. Любому другому приложению, которому необходимо обратиться к этой pod-оболочке, нужно будет хранить список адресов, что не кажется очень хорошей идеей.

К счастью, существует лучший способ: ресурс типа «сервис» предоставляет один несменяемый IP-адрес или такое же доменное имя, которые автоматически перенаправляют на любую подходящую pod-оболочку. Позже, в разделе «Ресурсы Ingress» на с. 226, мы поговорим о ресурсе Ingress, позволяющем выполнять более сложную маршрутизацию и использовать сертификаты TLS.

А пока поближе рассмотрим принцип работы сервисов в Kubernetes.

Сервис можно считать веб-прокси или балансировщиком нагрузки, который направляет запросы к группе *внутренних* pod-оболочек (рис. 4.2). Но он не ограничен веб-портами и может направить трафик на любой другой порт в соответствии с разделом `ports` в спецификации.

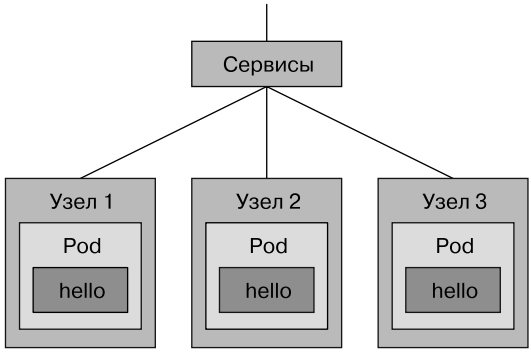

Рис. 4.2. Сервис предоставляет постоянную точку доступа к группе pod-оболочек

Ниже показан YAML-манифест сервиса для нашего демонстрационного приложения:

```
apiVersion: v1
kind: Service
metadata:
  name: demo
  labels:
    app: demo
spec:
```

```
ports:
  - port: 9999
  protocol: TCP
    targetPort: 8888
  selector:
    app: demo
  type: ClusterIP
```

Как видите, он чем-то напоминает ресурс развертывания, который мы показывали ранее. Однако параметр `kind` равен `Service`, а не `Deployment`, и раздел `spec` содержит только список портов, селектор и тип.

Если присмотреться поближе, можно заметить, что этот сервис перенаправляет свой порт 9999 к порту 8888 pod-оболочки:

```
...
ports:
- port: 9999
  protocol: TCP
  targetPort: 8888
```

Параметр `selector` говорит сервису, как перенаправлять запросы к конкретным pod-оболочкам. Чтобы получать запросы, pod-оболочка должна иметь подходящий набор меток: в данном случае это просто `app: demo` (см. раздел «Метки» на с. 205). В нашем примере есть только одна pod-оболочка, соответствующая этому критерию, но их могло бы быть несколько, и тогда сервис отправлял бы им запросы случайным образом[1].

В этом отношении сервисы Kubernetes немного похожи на традиционные балансировщики нагрузки: на самом деле объекты Service и Ingress могут автоматически создавать облачные балансировщики (см. раздел «Ресурсы Ingress» на с. 226).

Пока вам нужно запомнить только то, что развертывание управляет группой pod-оболочек вашего приложения, а сервис предоставляет запросам единую точку входа в эти pod-оболочки.

Теперь применим наш манифест, чтобы создать сервис:

```
kubectl apply -f k8s/service.yaml
service "demo" created

kubectl port-forward service/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

[1] Так работает алгоритм балансировки нагрузки по умолчанию; начиная с версии 1.10, Kubernetes поддерживает и другие алгоритмы, такие как минимум соединений. См. kubernetes.io/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive/.

Как и раньше, команда `kubectl port-forward` позволит подключить сервис `demo` к порту на вашем локальном компьютере, чтобы вы могли открыть адрес http://localhost:9999/ в своем браузере.

Убедитесь, что все работает корректно. Если вы удовлетворены, то прежде, чем переходить к следующему разделу, очистите каталог с помощью такой команды:

```
kubectl delete -f k8s/
```

Команду kubectl delete можно использовать с селектором label, как мы делали ранее: это позволяет удалять любые ресурсы, соответствующие этому селектору (см. раздел «Метки» на с. 205). В качестве альтернативы можно использовать представленную выше команду kubectl delete -f, указывая каталог с манифестами: все ресурсы, описанные в файлах манифестов, будут удалены.

УПРАЖНЕНИЕ

Отредактируйте файл `k8s/deployment.yaml`, поменяв количество реплик на 3. Примените манифест с помощью команды `kubectl apply` и убедитесь в том, что команда `kubectl get pods` теперь выводит три pod-оболочки `demo` вместо одной.

Обращение к кластеру с помощью kubectl

Утилиту `kubectl` можно сравнить со швейцарским армейским ножом для Kubernetes: она применяет конфигурацию, создает, модифицирует и уничтожает ресурсы, а также позволяет обращаться к кластеру за информацией о существующих ресурсах и их состоянии.

Мы уже видели, как с помощью `kubectl get` можно запрашивать pod-оболочки и развертывания. Эту команду также можно использовать для просмотра узлов вашего кластера:

```
kubectl get nodes
NAME                 STATUS   ROLES    AGE      VERSION
docker-for-desktop   Ready    master   1d       v1.10.0
```

Если вам нужно вывести ресурсы всех типов, выполните команду `kubectl get all` (на самом деле она возвращает не *все* типы ресурсов, а только самые распространенные, но на этом пока что мы не станем заострять внимание).

Чтобы получить исчерпывающую информацию об отдельной pod-оболочке (или любом другом ресурсе), выполните `kubectl describe`:

```
kubectl describe pod/demo-dev-6c96484c48-69vss
Name:           demo-dev-6c96484c48-69vss
Namespace:      default
Node:           docker-for-desktop/10.0.2.15
Start Time:     Wed, 06 Jun 2018 10:48:50 +0100
...
Containers:
  demo:
    Container ID:   docker://646aaf7c4baf6d...
    Image:          cloudnatived/demo:hello
...
Conditions:
  Type Status
  Initialized True
  Ready True
  PodScheduled True
...
Events:
  Type    Reason     Age   From               Message
  ----    ------     ---   ----               -------
  Normal  Scheduled  1d    default-scheduler  Successfully assigned demo-dev...
  Normal  Pulling    1d    kubelet            pulling image "cloudnatived/demo...
...
```

В примере вывода, показанном выше, мы видим, что `kubectl` возвращает некоторые основные сведения о самом контейнере, включая идентификатор его образа, состояние и упорядоченный список событий, которые произошли с этим контейнером (в главе 7 вы узнаете много всего о возможностях `kubectl`).

Выводим ресурсы на новый уровень

Теперь вы знаете все о том, что необходимо для развертывания приложений в кластерах Kubernetes с помощью декларативных манифестов в формате YAML. Однако эти файлы содержат много одинаковых фрагментов: например, вы несколько раз повторяете имя `demo`, селектор по метке `app: demo` и порт 8888.

Неужели эти значения нельзя указать один раз и затем ссылаться на них везде, где они встретятся в манифестах Kubernetes?

Например, было бы здорово иметь возможность определять переменные с названиями вроде `container.name` и `container.port` и затем использовать их в тех местах YAML-файлов, где они нужны. А если возникнет необходимость поменять

название приложения или номер порта, который оно прослушивает, отредактировать их лишь в одном месте, чтобы все манифесты обновились автоматически?

К счастью, для этого предусмотрен специальный инструмент, и в заключительном разделе этой главы мы продемонстрируем вам некоторые его возможности.

Helm: диспетчер пакетов для Kubernetes

Один из популярных диспетчеров пакетов для Kubernetes называется Helm. Он работает в точности так, как описано в предыдущем разделе. Вы можете использовать утилиту командной строки `helm` для установки и конфигурации приложений (собственных или чужих) и создавать пакеты, которые полностью описывают все необходимые для работы приложения ресурсы, включая их зависимости и настройки. Пакеты в Helm называются *чартами (charts)*.

Система Helm входит в семейство проектов Cloud Native Computing Foundation (см. раздел «Облачная ориентированность» на с. 44), что свидетельствует о ее стабильности и широком использовании.

Важно понимать, что Helm chart, в отличие от двоичных пакетов, которые используются такими инструментами, как APT или Yum, на самом деле не включает в себя образ контейнера как таковой. Вместо этого он содержит лишь метаданные о том, где образ можно найти, — точно так же, как при развертывании в Kubernetes.

Когда вы устанавливаете чарт, Kubernetes самостоятельно находит и загружает двоичный образ контейнера по указанному адресу. На самом деле это просто удобная обертка вокруг YAML-манифестов в Kubernetes.

Установка Helm

Следуйте инструкциям по установке Helm (helm.sh/docs/using_helm/#installing-helm) для вашей операционной системы.

Установив Helm, вы должны создать определенные ресурсы Kubernetes, чтобы открыть этому диспетчеру пакетов доступ к вашему кластеру. В каталоге `hello-helm` репозитория с примерами находится подходящий YAML-файл под названием `helm-auth.yaml`. Чтобы его применить, выполните следующие команды:

```
cd ../hello-helm
kubectl apply -f helm-auth.yaml
serviceaccount "tiller" created
clusterrolebinding.rbac.authorization.k8s.io "tiller" created
```

Установив необходимые права доступа, вы можете воспользоваться такой командой, чтобы инициализировать Helm для работы с кластером:

```
helm init --service-account tiller
$HELM_HOME has been configured at /Users/john/.helm.
```

Инициализация Helm может занять около пяти минут. Убедитесь в том, что все работает:

```
helm version
Client: &version.Version{SemVer:"v2.9.1",
GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.9.1",
GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710", GitTreeState:"clean"}
```

Если эта команда выполнилась успешно, вы готовы к работе с Helm. Если же появилось сообщение `Error: cannot connect to Tiller`, подождите несколько минут и попробуйте снова.

Установка чарта Helm

Как будет выглядеть чарт Helm для нашего приложения? В каталоге `hello-helm` можно заметить подкаталог `k8s`, который в предыдущем примере (`hello-k8s`) содержал только файлы манифеста Kubernetes для развертывания нашего кода. Теперь внутри него есть папка `demo` с чартом Helm внутри:

```
ls k8s/demo
Chart.yaml          prod-values.yaml staging-values.yaml    templates
values.yaml
```

В подразделе «Что внутри у чарта Helm» на с. 281 вы увидите, для чего нужны все эти файлы, ну а пока что воспользуемся Helm для установки нашего демонстрационного приложения. Для начала очистите ресурсы, оставшиеся от каких-либо предыдущих развертываний:

```
kubectl delete all --selector app=demo
```

Затем запустите следующую команду:

```
helm install --name demo ./k8s/demo
NAME:   demo
LAST DEPLOYED: Wed Jun  6 10:48:50 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
```

```
NAME              TYPE       CLUSTER-IP     EXTERNAL-IP  PORT(S)  AGE
demo-service      ClusterIP  10.98.231.112  <none>       80/TCP   0s

==> v1/Deployment
NAME      DESIRED  CURRENT   UP-TO-DATE   AVAILABLE   AGE
demo      1        1         1            0           0s

==> v1/Pod(related)
NAME                       READY   STATUS              RESTARTS  AGE
demo-6c96484c48-69vss      0/1     ContainerCreating   0         0s
```

Вы видите, что диспетчер Helm создал ресурс Deployment (который запускает pod-оболочку) и Service, как и в предыдущем примере. Для этого команда `helm install` создает объект Kubernetes под названием *«выпуск» (release)*.

Чарты, репозитории и выпуски

Это самые важные термины, относящиеся к Helm, которые вам нужно знать.

- *Чарт* — пакет Helm с определениями всех ресурсов, необходимых для выполнения приложения в Kubernetes.
- *Репозиторий* — место, где можно делиться своими чартами и загружать чужие.
- *Выпуск* — конкретный экземпляр чарта, запущенный в кластере Kubernetes.

Один чарт можно устанавливать несколько раз в одном и том же кластере. Например, у вас может быть запущено несколько копий чарта с веб-сервером Nginx, каждая из которых обслуживает свой сайт. Один экземпляр чарта является отдельным выпуском.

У каждого выпуска есть уникальное имя, которое можно указать в команде `helm install` с помощью флага `-name` (если не указать имя, Helm выберет случайный выпуск, что вряд ли вам понравится).

Вывод списка выпусков Helm

Чтобы проверить, какие выпуски запущены на данный момент, выполните команду `helm list`:

```
helm list
NAME    REVISION UPDATED                  STATUS   CHART      NAMESPACE
demo    1        Wed Jun   6 10:48:50 2018 DEPLOYED demo-1.0.1 default
```

Чтобы вывести состояние конкретного выпуска, передайте его имя команде `helm status`. Вы увидите ту же информацию, что и при первом развертывании выпуска:

список всех ресурсов Kubernetes, которые с ним связаны, включая развертывания, pod-оболочки и сервисы.

Позже в этой книге мы покажем, как создавать чарты Helm для собственных приложений (см. подраздел «Что внутри у чарта Helm» на с. 281). А пока что вам достаточно воспринимать Helm как удобный способ установки публичных чартов.

Полный список публичных чартов Helm (github.com/helm/charts/tree/master/stable) можно просмотреть в GitHub.

Список доступных чартов также можно получить, выполнив команду helm search без аргументов (или helm search redis, если вам, к примеру, нужно найти чарт для Redis).

Резюме

Эта книга не о внутренних механизмах Kubernetes (деньги, извините, не возвращаем). Наша цель — показать *возможности* этой платформы и быстро подготовить вас к запуску реальных рабочих заданий в промышленных условиях. Тем не менее мы считаем полезным дать вам представление об отдельных основных внутренних компонентах, с которыми вы будете работать, например о pod-оболочках и развертываниях. В этой главе мы кратко обсудили некоторые наиболее важные из них.

Какими бы увлекательными ни казались технические детали нам, гикам, мы все равно заинтересованы в достижении результата. Поэтому мы воздержались от исчерпывающего рассмотрения каждого вида ресурсов, которые предоставляет Kubernetes: их довольно *много* и значительная часть вам точно не нужна (по крайней мере, не сейчас).

На данном этапе, мы думаем, вам необходимо знать следующее.

- Pod-оболочка — это фундаментальная единица работы в Kubernetes, которая определяет один или несколько контейнеров, взаимодействующих между собой и запланированных к одновременному выполнению.
- Развертывание — высокоуровневый ресурс Kubernetes, который декларативным образом управляет pod-оболочками, их доставкой, планированием, обновлением и перезапуском, когда это необходимо.
- Сервис в Kubernetes — это эквивалент балансировщика нагрузки или прокси, который направляет трафик к подходящим pod-оболочкам через единый, публичный и долгосрочный IP-адрес или доменное имя.

- Планировщик Kubernetes ищет pod-оболочки, которые еще нигде не выполняются, находит для них подходящие узлы и приказывает утилите kubelet запустить эти pod-оболочки на найденных узлах.
- Такие ресурсы, как развертывания, представлены записями во внутренней базе данных Kubernetes. Внешне эти ресурсы могут иметь форму текстовых файлов (известных как *манифесты*) в формате YAML. Манифест — это объявление желаемого состояния ресурса.
- `kubectl` — основной инструмент для взаимодействия с Kubernetes, который позволяет применять манифесты, запрашивать данные о ресурсах, вносить изменения, удалять ресурсы и многое другое.
- Helm — это диспетчер пакетов для Kubernetes. Он упрощает конфигурацию и развертывание приложений, позволяя генерировать YAML-файлы с помощью группы шаблонов и единого набора значений (таких как имя приложения или порт, который оно прослушивает), вместо того чтобы писать эти файлы вручную.

5 Управление ресурсами

> Кому недостаточно малого, тому ничего не достаточно.
>
> *Эпикур*

Из этой главы вы узнаете, как использовать кластер максимально эффективно: мы покажем, как контролировать и оптимизировать потребление ресурсов, как управлять жизненным циклом контейнеров и как разделить кластер с помощью пространств имен. Мы также поговорим об отдельных методиках и приемах, которые позволят снизить стоимость вашего кластера и достичь наилучшего соотношения цены и качества.

Вы научитесь использовать запросы к ресурсам, лимиты и значения по умолчанию, увидите, как их оптимизировать с помощью вертикального автомасштабирования pod-оболочек. Мы покажем, как проверять готовность и работоспособность и использовать ресурс `PodDisruptionBudget` для управления контейнерами, как оптимизировать облачное хранилище, как и когда применять прерываемые или зарезервированные серверы для контроля за расходами.

Понимание ресурсов

Представьте, что у вас есть кластер Kubernetes заданной мощности с разумным количеством узлов подходящего размера. Как получить от него наилучший результат с учетом потраченных денег? То есть как добиться того, чтобы ваши рабочие задания использовали доступные ресурсы кластера по максимуму, но при этом оставалось достаточно пространства для маневра на случай скачков нагрузки, отказа узлов и неудачных развертываний?

Чтобы ответить на этот вопрос, поставьте себя на место планировщика Kubernetes и попытайтесь взглянуть на ситуацию с его точки зрения. Задача планировщика состоит в выборе места для выполнения той или иной pod-оболочки: есть ли у вас узлы с достаточным объемом свободных ресурсов?

На этот вопрос можно ответить только в том случае, если планировщик знает, сколько ресурсов потребуется pod-оболочке для работы. Если ей нужен 1 ГиБ памяти, ее выполнение нельзя запланировать на узле, свободная память которого равна лишь 100 МиБ.

Точно так же у планировщика должна быть возможность предпринять определенные действия, если pod-оболочка начала «жадничать» и отхватила слишком много ресурсов, необходимых ее соседям по узлу. Но что означает «слишком много»? Для эффективной работы планировщик должен знать минимальные и максимальные требования к ресурсам для каждой pod-оболочки.

Здесь на помощь приходят запросы и лимиты на ресурсы в Kubernetes. Система сама понимает, как обращаться с двумя видами ресурсов: центральным процессором и памятью. Бывают и другие важные характеристики, такие как пропускная способность сети, скорость дисковых операций ввода/вывода и размер дискового пространства. Из-за них в кластере может возникать конкуренция, но Kubernetes пока не позволяет указывать такие ресурсы среди требований pod-оболочек.

Единицы измерения ресурсов

Использование процессорных ресурсов pod-оболочкой выражается, как можно было ожидать, в единицах процессоров. Одна такая единица эквивалентна одному AWS vCPU, одному Google Cloud Core, одному Azure vCore и одному *гиперпотоку* физического процессора, который поддерживает гиперпоточность. Иными словами, *1 CPU* с точки зрения Kubernetes означает то, о чем можно было бы подумать.

Поскольку большинству pod-оболочек не нужен целый процессор, запросы и лимиты обычно выражаются в *миллипроцессорах* (которые иногда называют *миллиядрами*). Память измеряется в байтах или, что более удобно, в *мебибайтах* (МиБ).

Запросы ресурсов

Запрос ресурса в Kubernetes определяет минимальный объем этого ресурса, который необходим для работы pod-оболочки. Например, запрос `100m` (100 миллипроцессоров) и `250Mi` (250 МиБ памяти) означает, что pod-оболочка не может быть назначена узлу с меньшим количеством доступных ресурсов. Если в кластере нет ни одного узла с достаточной мощностью, pod-оболочка будет оставаться в состоянии `pending`, пока такой узел не появится.

Например, если все узлы вашего кластера имеют два процессорных ядра и 4 ГиБ памяти, то контейнер, запрашивающий 2,5 процессора или 5 ГиБ памяти, никогда не будет запланирован.

Посмотрим, как бы выглядели запросы ресурсов в нашем демонстрационном приложении:

```
spec:
   containers:
   - name: demo
     image: cloudnatived/demo:hello
     ports:
     - containerPort: 8888
     resources:
       requests:
         memory: "10Mi"
         cpu: "100m"
```

Лимиты на ресурсы

Лимит на ресурс определяет максимальное количество этого ресурса, которое pod-оболочке позволено использовать. Если pod-оболочка попытается занять больше выделенного ей лимита на процессор, производительность будет снижена.

Pod-оболочка, пытающаяся использовать больше, чем указано в лимите на память, будет принудительно остановлена, и ее выполнение, если это возможно, снова окажется запланировано. На практике это означает, что pod-оболочка может перезапуститься на том же узле.

Некоторые приложения, такие как сетевые серверы, могут со временем потреблять все больше и больше ресурсов в ответ на растущую нагрузку. Задание лимитов на ресурсы — это хороший способ не давать «прожорливым» pod-оболочкам потреблять больше той доли ресурсов, которая им полагается.

Вот пример задания лимита на ресурсы для нашего демонстрационного приложения:

```
spec:
   containers:
   - name: demo
     image: cloudnatived/demo:hello
     ports:
     - containerPort: 8888
     resources:
       limits:
         memory: "20Mi"
         cpu: "250m"
```

То, какие лимиты следует устанавливать для конкретного приложения, зависит от ваших наблюдений и личного мнения (см. подраздел «Оптимизация pod-оболочек» на с. 123).

Kubernetes допускает *отрицательный баланс ресурсов*, когда сумма всех лимитов у контейнеров одного узла превышает общее количество ресурсов, которыми узел обладает. Это своего рода азартная игра: планировщик ставит на то, что большинство контейнеров большую часть времени не будут достигать своих лимитов.

Если ставка себя не оправдает, общее количество потребляемых ресурсов начнет приближаться к максимальной мощности узла и Kubernetes начнет удалять контейнеры более агрессивно. В условиях нехватки ресурсов могут быть остановлены даже те контейнеры, которые исчерпали запрошенные ресурсы, а не лимиты[1].

При прочих равных, когда возникает необходимость в удалении pod-оболочек, Kubernetes начинает с тех, которые больше всего превысили запрошенные ресурсы. Pod-оболочки, выходящие за эти рамки, удаляются только в очень редких ситуациях: если без этого действия не хватит ресурсов для системных компонентов Kubernetes, таких как `kubelet`.

Рекомендованный подход

Всегда заказывайте запросы и лимиты на ресурсы для своих контейнеров. Это поможет Kubernetes как следует управлять вашими pod-оболочками и планировать их работу.

Делайте контейнеры небольшими

В подразделе «Минимальные образы контейнеров» на с. 56 мы упоминали, что создание контейнеров как можно меньшего размера является хорошей идеей. Тому есть множество причин.

- Мелкие контейнеры быстрее собираются.
- Образы занимают меньше места.
- Загрузка образов с сервера происходит быстрее.
- Уменьшается поверхность атаки.

[1] Это поведение можно регулировать для отдельных контейнеров, используя классы качества обслуживания (Quality of Service, QoS), kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/.

Если вы используете Go, у вас уже есть фора, поскольку Go может скомпилировать ваше приложение в единый статически скомпонованный двоичный файл. Если в вашем контейнере всего один файл — меньше уже не придумаешь!

Управление жизненным циклом контейнера

Мы уже могли убедиться в том, что платформа Kubernetes лучше всего управляет pod-оболочками тогда, когда знает их требования к процессору и памяти. Но она также должна знать, когда контейнер работает: то есть когда он правильно функционирует и готов к обработке запросов.

Контейнеризированные приложения довольно часто входят в ступор: их процесс все еще выполняется, но больше не обслуживает никакие запросы. Kubernetes нужен какой-то способ для обнаружения подобных ситуаций, чтобы решить проблему за счет перезапуска контейнера.

Проверки работоспособности

Kubernetes позволяет задать проверку *работоспособности* в рамках спецификации контейнера. Она будет определять, жив ли контейнер (то есть работает ли он).

Для контейнера с HTTP-сервером определение проверки работоспособности обычно выглядит примерно так:

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8888
  initialDelaySeconds: 3
  periodSeconds: 3
```

Проверка `httpGet` делает HTTP-запрос к URI и порту, которые вы указали: в данном случае это `/healthz` на порте 8888.

Если у вашего приложения нет отдельной конечной точки для такой проверки, можете использовать символ `/` или любой корректный URL-адрес. Хотя обычно для этих целей специально создается конечная точка `/healthz`. Почему `z` в конце? Просто чтобы случайно не создать конфликт с существующим путем вроде `health`, который, к примеру, может вести на страницу с медицинской информацией.

Если приложение отвечает с помощью HTTP-кода вида 2xx или 3xx, Kubernetes считает его активным. Любой другой ответ или отсутствие такового говорит о том, что контейнер отключен и его нужно перезапустить.

Задержка и частота проверки

Как скоро Kubernetes следует начинать проверять работоспособность? Никакое приложение не может стартовать мгновенно. Если проверка произойдет сразу после запуска контейнера, она, скорее всего, потерпит неудачу: в результате контейнер перезапустится — и этот цикл будет повторяться бесконечно!

Поле `initialDelaySeconds` позволяет указать время ожидания перед первой проверкой работоспособности, чтобы избежать *убийственного цикла (loop of the death)*.

Точно так же Kubernetes лучше не заваливать ваше приложение запросами к конечной точке `healthz` тысячу раз в секунду. Поле `periodSeconds` определяет, как часто следует выполнять проверку работоспособности: в данном примере это делается каждые три секунды.

Другие типы проверок

Помимо `httpGet`, доступны и другие типы проверок. Для сетевых серверов, которые не понимают HTTP, можно использовать `tcpSocket`:

```
livenessProbe:
  tcpSocket:
    port: 8888
```

Если TCP-соединение с заданным портом будет успешно установлено, контейнер жив.

Вы можете также выполнять в контейнере произвольные команды, используя проверку `exec`:

```
readinessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
```

Проверка `exec` выполняет внутри контейнера заданную команду и считается успешной при успешном выполнении команды (то есть если завершается с нулевым статусом). Обычно `exec` лучше подходит для проверки готовности.

Проверки gRPC

Многие приложения и сервисы общаются по HTTP, но все большую популярность, особенно в мире микросервисов, набирает двоичный сетевой протокол gRPC (grpc.io), отличающийся своей эффективностью и переносимостью. Он был разработан компанией Google и развивается под эгидой Cloud Native Computing Foundation.

Проверки `httpGet` не работают с серверами gRPC. Конечно, вместо этого можно использовать `tcpSocket`, но в таком случае вы будете знать только о факте подключения к сокету, что ничего не скажет вам о состоянии самого сервера.

У gRPC есть стандартный механизм проверки работоспособности с поддержкой большинства gRPC-сервисов. Для обращения к нему из Kubernetes можно воспользоваться инструментом grpc-health-probe (kubernetes.io/blog/2018/10/01/health-checking-grpc-servers-on-kubernetes).

Проверки готовности

Проверки *готовности* и работоспособности имеют общее происхождение, но разную семантику. Иногда приложению нужно просигнализировать Kubernetes о том, что оно временно неспособно обрабатывать запросы: возможно, выполняет длинный процесс инициализации или ждет завершения какого-то дочернего процесса. Для этого предусмотрена проверка готовности.

Если ваше приложение не начинает прослушивать HTTP, пока не будет готово к обработке запросов, проверки готовности и работоспособности могут выглядеть одинаково:

```
readinessProbe:
  httpGet:
    path: /healthz
    port: 8888
  initialDelaySeconds: 3
  periodSeconds: 3
```

Контейнер, не прошедший проверку готовности, удаляется из любых сервисов, совпавших с заданной pod-оболочкой. Это похоже на удаление неисправного узла из пула балансировщика нагрузки: к pod-оболочке не будет направляться трафик, пока она опять не начнет успешно проходить проверку готовности.

Обычно при запуске pod-оболочки Kubernetes начинает отправлять ей трафик, как только контейнер переходит в рабочее состояние. Но если у контейнера есть проверка готовности, Kubernetes сначала дождется ее успешного прохождения и только потом начнет отправлять какие-либо запросы: благодаря этому пользователи

не увидят ошибок от неготовых контейнеров. Это критически важно для обновлений без простоя (больше об этом — в разделе «Стратегии развертывания» на с. 302).

Контейнер, который еще не готов, все равно имеет состояние `Running`, но в поле `READY` будет видно, что один или несколько контейнеров в его pod-оболочке не готовы к работе:

```
kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
readiness-test   0/1     Running   0          56s
```

Проверки готовности должны возвращать только HTTP-статус 200 OK. И хотя сама платформа Kubernetes считает, что о готовности сигнализируют коды вида 2xx и 3xx, у облачного балансировщика нагрузки может быть другое мнение. Если ваш ресурс Ingress работает совместно с облачным балансировщиком нагрузки (см. раздел «Ресурсы Ingress» на с. 226) и ваша проверка готовности, к примеру, возвращает код перенаправления 301, балансировщик может пометить все pod-оболочки как неработоспособные. Убедитесь в том, что проверки готовности возвращают только код состояния 200.

Проверки готовности на основе файла

В качестве альтернативы вы могли бы заставить приложение создать в файловой системе контейнера файл с названием вроде `/tmp/healthy` и использовать проверку готовности `exec` для того, чтобы удостовериться в наличии этого файла.

Проверка готовности такого рода может быть полезной. Например, если нужно временно выключить контейнер с целью отладки, к нему можно подключиться и удалить файл `/tmp/healthy`. Следующая проверка готовности будет неудачной, и Kubernetes уберет контейнер из списка подходящих сервисов (хотя для этого лучше подправить метки контейнера, чтобы он больше не соответствовал своему сервису: см. подраздел «Ресурсы типа “сервис”» на с. 97).

Теперь вы можете спокойно проинспектировать и отладить данный контейнер. Когда закончите, можете его удалить и развернуть исправленную версию или вернуть на место проверочный файл, чтобы контейнер опять начал получать трафик.

Рекомендуемый подход

Используйте проверки готовности и работоспособности, чтобы сообщать Kubernetes о том, готово ли ваше приложение обрабатывать запросы и требует ли оно перезапуска из-за какой-то проблемы.

Поле minReadySeconds

По умолчанию контейнеры или pod-оболочки считаются готовыми к работе на момент успешного прохождения проверки готовности. В некоторых случаях контейнеру необходимо дать поработать какое-то время, чтобы убедиться в его стабильности. Kubernetes во время развертывания ждет готовности каждой новой pod-оболочки, прежде чем запускать следующую (см. подраздел «Плавающие обновления RollingUpdate» на с. 303). Если неисправный контейнер сразу же отказывает, это останавливает выкатывание обновлений, но если сбой происходит через несколько секунд, до обнаружения проблемы все его реплики могут успеть выкатиться.

Чтобы этого избежать, для контейнера можно установить поле `minReadySeconds`. Контейнеры или pod-оболочки не будут считаться готовыми, пока с момента успешной проверки готовности не пройдет `minReadySeconds` секунд (по умолчанию 0).

Ресурс PodDisruptionBudget

Иногда Kubernetes нужно остановить ваши pod-оболочки, даже если они в полном порядке и готовы к работе (этот процесс называется *выселением*). Возможно, узел, на котором они размещены, очищается перед обновлением и pod-оболочки необходимо переместить на другой узел.

Но это вовсе не должно приводить к простаиванию вашего приложения при условии, что вы можете запустить достаточное количество реплик. Ресурс `PodDisruptionBudget` позволяет указать, сколько pod-оболочек заданного приложения допустимо к потере в любой момент времени.

Например, вы можете указать, что ваше приложение допускает одновременное нарушение работы не более чем 10 % pod-оболочек. При желании Kubernetes можно разрешить выселять любое количество pod-оболочек при условии, что всегда будет оставаться как минимум три рабочие реплики.

minAvailable

Ниже показан пример ресурса `PodDisruptionBudget`, который использует поле `minAvailable` для определения минимального количества рабочих pod-оболочек:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: demo-pdb
```

```
spec:
  minAvailable: 3
  selector:
    matchLabels:
      app: demo
```

Значение `minAvailable: 3` говорит о том, что в рабочем состоянии должны находиться как минимум три pod-оболочки, соответствующие метке `app: demo`. Kubernetes может выселить любое количество pod-оболочек `demo`, но обязательно оставить три из них.

maxUnavailable

С другой стороны, поле `maxUnavailable` ограничивает общее количество или процент pod-оболочек, которые Kubernetes позволено выселить:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: demo-pdb
spec:
  maxUnavailable: 10%
  selector:
  matchLabels:
    app: demo
```

В соответствии с этими параметрами в любой момент времени можно выселить не более чем 10 % всех pod-оболочек `demo`. Но это относится лишь к так называемому *добровольному выселению*, то есть инициированному системой Kubernetes. Если, к примеру, узел испытывает аппаратные неполадки или удаляется, его pod-оболочки выселяются принудительно, даже если это нарушает параметры `PodDisruptionBudget`.

Kubernetes пытается равномерно распределить pod-оболочки по узлам, поэтому при прочих равных условиях это следует учитывать в ходе вычисления количества узлов, необходимых для вашего кластера. Если у вас есть три узла, отказ одного из них может привести к потере трети всех ваших pod-оболочек, и в итоге может не остаться ресурсов для поддержания приемлемого уровня обслуживания (см. подраздел «Высокая доступность» на с. 67).

Рекомендованный подход

Устанавливайте PodDisruptionBudget для приложений, которые являются критически важными для бизнеса. Благодаря этому у вас всегда будет достаточно реплик для продолжения обслуживания, даже в случае выселения pod-оболочек.

Использование пространств имен

Еще одним полезным механизмом контроля за потреблением ресурсов в вашем кластере является использование *пространств имен*. Пространство имен в Kubernetes предоставляет способ разделения кластера на отдельные части в тех или иных целях.

Например, у вас может быть пространство имен `prod` для промышленных приложений и `test` — для экспериментов. Как предполагает термин «пространство имен», имена одного пространства не видны в другом.

Это означает, что в пространствах имен `prod` и `test` у вас может быть два разных сервиса с именем `demo`, которые не будут между собой конфликтовать.

Чтобы увидеть, какие пространства имен существуют в вашем кластере, выполните следующую команду:

```
kubectl get namespaces
NAME          STATUS   AGE
default       Active   1y
kube-public   Active   1y
kube-system   Active   1y
```

Пространства имен можно представить себе в виде папок на жестком диске вашего компьютера. *Теоретически* все файлы можно хранить в одной папке, но это неудобно: поиск конкретного файла занимал бы много времени и было бы сложно понять, какие файлы связаны друг с другом. Пространство имен группирует связанные между собой ресурсы и упрощает работу с ними. Хотя, в отличие от папок, пространства имен не могут быть вложенными.

Работа с пространствами имен

До сих пор при работе с Kubernetes мы всегда использовали *пространство имен по умолчанию*. Оно выбирается автоматически, если при работе с командой `kubectl` (например, `kubectl run`) не указать другое пространство. Выше вы могли заметить название `kube-system`. Если вам интересно, это пространство имен, в котором выполняются внутренние компоненты Kubernetes. Благодаря этому они отделены от приложений.

Ваша команда будет использовать то пространство имен, которое вы укажете с помощью флага `--namespace` (сокращенно `-n`). Например, чтобы получить список pod-оболочек в пространстве имен `prod`, выполните:

```
kubectl get pods --namespace prod
```

Какое пространство имен следует использовать

Вы сами решаете, как разделить свой кластер на пространства имен. Один из интуитивно понятных подходов заключается в том, что отдельное пространство имен выделяется для каждого приложения или команды. Например, вы можете создать пространство `demo` для выполнения в нем одноименного приложения. В Kubernetes пространства имен можно создавать с помощью ресурса Namespace, как показано ниже:

```
apiVersion: v1
kind: Namespace
metadata:
  name: demo
```

Чтобы применить манифест ресурса, используйте команду `kubectl apply -f` (подробнее об этом — в разделе «Манифесты ресурсов в формате YAML» на с. 95). YAML-манифесты для всех примеров в этом разделе можно найти в каталоге `hello-namespace` репозитория приложения `demo`:

```
cd demo/hello-namespace
ls k8s
deployment.yaml   limitrange.yaml   namespace.yaml   resourcequota.yaml
service.yaml
```

Вы можете пойти дальше и создать по отдельному пространству имен для каждой среды, в которой работает ваше приложение: например, `demo-prod`, `demo-staging`, `demo-test` и т. д. Пространство имен можно использовать в качестве временного *виртуального кластера*, который, став ненужным, удаляется. Но будьте осторожны! Вместе с пространством имен удаляются все его внутренние ресурсы. Убедитесь в том, что выбрали правильное пространство (в подразделе «Введение в управление доступом на основе ролей» на с. 258 объясняется, как выдавать или забирать права доступа к отдельным пространствам имен).

В текущей версии Kubernetes нет способа *защиты* ресурсов, включая пространства имен, от удаления (хотя предложение (github.com/kubernetes/kubernetes/issues/10179) о добавлении такой возможности уже обсуждается). Поэтому удаляйте пространства имен, только если они действительно временные и не содержат никаких промышленных ресурсов.

Рекомендованный подход

Создавайте отдельные пространства имен для каждого приложения или логического компонента своей инфраструктуры. Не используйте пространство имен по умолчанию: так слишком легко ошибиться.

Если вам нужно заблокировать весь сетевой трафик — входящий или исходящий — в конкретном пространстве имен, это можно сделать с помощью сетевых политик Kubernetes (kubernetes.io/docs/concepts/services-networking/network-policies).

Адреса сервисов

Несмотря на то что пространства имен изолированы друг от друга, они по-прежнему могут взаимодействовать с чужими сервисами. Как вы помните по подразделу «Ресурсы типа “сервис”» на с. 97, у каждого сервиса в Kubernetes есть доменное имя, через которое с ним можно общаться. Обратившись по сетевому имени `demo`, вы подключитесь к одноименному сервису. Как это работает между разными пространствами имен?

Доменные имена сервисов всегда имеют следующую структуру:

```
SERVICE.NAMESPACE.svc.cluster.local
```

Приставка `.svc.cluster.local` является необязательной, равно как и пространство имен. Но если, например, вы хотите обратиться к сервису `demo` из пространства `prod`, можно использовать такое имя:

```
demo.prod
```

Даже если у вас есть десять разных сервисов под названием `demo`, каждый из которых размещен в отдельном пространстве, вы можете уточнить, какой из них имеется в виду, добавив к доменному имени пространство имен.

Квоты на ресурсы

Вы можете ограничить потребление процессорного времени и памяти не только для отдельных контейнеров, как было показано в подразделе «Запросы ресурсов» на с. 108, но и для заданных пространств имен. Для этого в соответствующем пространстве нужно создать ресурс `ResourceQuota`. Ниже показан пример:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: demo-resourcequota
spec:
  hard:
    pods: "100"
```

Применение этого манифеста к конкретному пространству имен (например, к `demo`) устанавливает жесткий лимит на запуск не более чем 100 pod-оболочек в этом

пространстве (следует отметить, что параметр `metadata.name` внутри `ResourceQuota` может быть каким угодно: затрагиваются лишь те пространства, к которым вы применяете данный манифест):

```
cd demo/hello-namespace
kubectl create namespace demo
namespace "demo" created
kubectl apply --namespace demo -f k8s/resourcequota.yaml
resourcequota "demo-resourcequota" created
```

Теперь Kubernetes будет блокировать любые API-операции в пространстве имен `demo`, которые превышают квоту. В этом примере `ResourceQuota` ограничивает пространство имен 100 pod-оболочками, поэтому при попытке запуска 101-й pod-оболочки вы увидите такое сообщение об ошибке:

```
Error from server (Forbidden): pods "demo" is forbidden: exceeded quota:
demo-resourcequota, requested: pods=1, used: pods=100, limited: pods=100
```

Использование `ResourceQuota` — это хороший способ не дать приложениям из одного пространства имен захватить слишком много ресурсов у других частей кластера.

Вы также можете ограничить общее количество ресурсов процессора и памяти, которые используют pod-оболочки в одном пространстве имен, но мы не советуем этого делать. Если задать слишком низкий лимит, то при приближении к нему ваши рабочие задания, скорее всего, столкнутся с неожиданными и завуалированными проблемами. Если лимит окажется слишком высоким, он потеряет всякий смысл.

Тем не менее такой лимит поможет предотвратить генерацию практически неограниченного количества pod-оболочек в результате неправильной конфигурации или опечаток. Очень легко забыть об очистке объекта от какого-то регулярного задания и в один прекрасный момент обнаружить, что подобных заданий у вас теперь тысячи и все они загромождают кластер.

Рекомендуемый подход

Используйте ResourceQuotas в каждом пространстве имен, чтобы ограничить количество pod-оболочек, которые могут в нем выполняться.

Чтобы проверить, активирован ли ресурс `ResourceQuotas` в конкретном пространстве имен, используйте команду `kubectl get resourcequotas`:

```
kubectl get resourcequotas -n demo
NAME                 AGE
demo-resourcequota   15d
```

Запросы и лимиты на ресурсы по умолчанию

Не всегда заранее понятно, какие требования к ресурсам будут у вашего контейнера. Можно установить запросы и лимиты на ресурсы по умолчанию для всех контейнеров в пространстве имен, используя манифест `LimitRange`:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: demo-limitrange
spec:
  limits:
  - default:
      cpu: "500m"
      memory: "256Mi"
    defaultRequest:
      cpu: "200m"
      memory: "128Mi"
    type: Container
```

Как и в случае с ResourceQuotas, параметр metadata.name в LimitRange может быть каким угодно. Может, например, не соответствовать пространству имен Kubernetes. Ресурсы LimitRange и ResourceQuota начинают действовать только после того, как их применили к определенному пространству имен.

Если в пространстве имен нет запросов или лимитов на ресурсы, любой его контейнер унаследует значение по умолчанию от `LimitRange`. Например, контейнер, для которого не указан запрос `cpu`, наследует у `LimitRange` показатель `200m`. Аналогичным образом контейнер без лимита `memory` унаследует у `LimitRange` показатель `256Mi`.

Таким образом, теоретически можно задать значения по умолчанию в `LimitRange` и не тратить время на прописывание запросов и лимитов для отдельных контейнеров. Однако это плохой подход: вы должны иметь возможность взглянуть на спецификацию контейнера и понять, какие у него запросы и лимиты, без знания о том, применяется ли `LimitRange`. Используйте `LimitRange` только в качестве меры предосторожности на случай, если владельцы контейнеров забыли указать запросы и лимиты.

Рекомендуемый подход

Используйте LimitRanges в каждом пространстве имен, чтобы установить запросы и лимиты на ресурсы контейнеров по умолчанию, но не полагайтесь на них, а воспринимайте как меру предосторожности. Всегда явно прописывайте запросы и лимиты в спецификации самого контейнера.

Оптимизация стоимости кластера

В разделе «Масштабирование и изменение размеров кластера» на с. 137 мы высказали некоторые соображения относительно выбора начального размера для вашего кластера и его масштабирования по мере изменения рабочих нагрузок. Но если предположить, что кластер имеет подходящий размер и достаточную мощность, как максимально оптимизировать денежные расходы на его работу?

Оптимизация развертываний

Действительно ли вам нужно столько реплик? Это выглядит констатацией очевидного факта, но ресурсы, которые потребляет каждая pod-оболочка в кластере, недоступны какой-то другой pod-оболочке.

Запуск большого количества реплик для всего подряд может показаться заманчивым, так как это гарантирует, что при отказе отдельных pod-оболочек или выполнении плавающих обновлений качество обслуживания никогда не пострадает. К тому же чем больше у вас реплик, тем с большим объемом трафика может справиться ваше приложение.

Однако реплики следует использовать — кластер способен выполнять ограниченное количество pod-оболочек. Выделите их для приложений, которым действительно нужны максимальные доступность и производительность.

Если вам не так уж и важно, что определенное развертывание станет недоступным на несколько секунд во время обновления, это означает, что ему не нужно много реплик: на удивление много приложений и сервисов могут прекрасно работать с одной или двумя.

Посмотрите, сколько реплик указано в конфигурации для каждого развертывания, и спросите себя:

- какие бизнес-требования к производительности и доступности у этого сервиса;
- можем ли мы удовлетворить эти требования меньшим числом реплик?

Если приложение не справляется с нагрузкой или слишком много пользователей получают ошибки во время обновления развертывания, это означает, что нужно больше реплик. Но часто размер развертывания можно существенно сократить еще до того, как снижение качества работы станет заметным.

Рекомендуемый подход

Используйте для заданного развертывания то минимальное количество pod-оболочек, которого будет достаточно для удовлетворения ваших требований к производительности и доступности. Постепенно уменьшайте количество реплик до момента, когда задачи уровня сервиса едва выполняются.

Оптимизация pod-оболочек

Ранее в этой главе, в подразделе «Запросы ресурсов» на с. 108, мы подчеркивали важность установки корректных запросов и лимитов на ресурсы для ваших контейнеров. Если сделать запросы слишком низкими, вскоре это даст о себе знать: pod-оболочки начнут отказывать. А если слишком высокими, может случиться так, что об этом вы узнаете только при получении счета за месяц от своего облачного провайдера.

Вы должны регулярно следить за запросами и лимитами на ресурсы для различных рабочих задач и сравнивать их с тем, сколько ресурсов использовали на самом деле.

Большинство сервисов по управлению Kubernetes предлагают некую приборную панель, которая выводит историю использования памяти и процессорного времени вашими контейнерами (подробнее об этом — в разделе «Мониторинг состояния кластера» на с. 271).

Вы также можете создать свою собственную панель со статистикой с помощью Prometheus и Grafana (об этом речь пойдет в главе 15).

Задание оптимальных запросов и лимитов на ресурсы сродни искусству, и результат будет разным для каждого вида рабочей нагрузки. Некоторые контейнеры могут простаивать большую часть времени, изредка демонстрируя всплески потребления ресурсов при обработке запроса, другие могут быть постоянно заняты и постепенно повышать потребление памяти, пока не достигнут своих лимитов.

В целом лимиты на ресурсы следует устанавливать чуть выше того максимума, который контейнер потребляет в нормальном режиме. Например, если потребление памяти заданного контейнера на протяжении нескольких дней не превышает 500 МиБ, лимит на память можно указать в размере 600 МиБ.

Должны ли контейнеры вообще иметь лимиты? Кто-то считает, что в промышленных условиях не должно быть ограничений или что лимиты следует делать такими высокими, чтобы контейнеры их никогда не достигали. Это может иметь смысл для крупных и ресурсоемких контейнеров, но мы считаем, что ограничения нужны в любом случае. Без них контейнер с утечкой памяти или слишком высокими требованиями к процессору может израсходовать все доступные ресурсы на узле и затруднить работу своих соседей.

Чтобы избежать такой ситуации, назначьте контейнеру лимиты, немного превышающие 100 % от уровня обычного потребления. Благодаря этому контейнер не будет удален в условиях нормальной работы, а если что-то пойдет не так, масштабы последствий будут минимизированы.

Значения запросов менее критичны по сравнению с лимитами, но их тоже нельзя устанавливать слишком высокими (так как работа pod-оболочки никогда не будет запланирована) или слишком низкими (так как pod-оболочки, превысившие свои запросы, первые в очереди на выселение).

Vertical Pod Autoscaler

У Kubernetes есть дополнение под названием Vertical Pod Autoscaler (VPA) (github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler), которое может помочь вам в подборе идеальных значений для запросов ресурсов. Оно следит за заданным развертыванием и автоматически регулирует запросы ресурсов для его pod-оболочек, исходя из того, что те на самом деле используют. У дополнения есть пробный режим, который просто дает советы, не модифицируя запущенные pod-оболочки, и это может быть полезно.

Оптимизация узлов

Размер узлов, на которых может работать Kubernetes, довольно сильно варьируется, но некоторые размеры подходят лучше других. Чтобы получить оптимальную мощность кластера за свои деньги, вам нужно понаблюдать за работой узлов в реальных условиях и с вашими конкретными рабочими нагрузками. Это поможет определить самые экономически выгодные типы серверов.

Стоит помнить о том, что у каждого узла есть операционная система, которая потребляет ресурсы диска, памяти и процессора: равно как их потребляют и системные компоненты Kubernetes, и среда выполнения контейнеров. Чем меньше узел, тем бóльшая доля его ресурсов отводится под эти накладные расходы.

Таким образом, более крупные узлы могут быть более рентабельными, поскольку доля их ресурсов, доступных для ваших рабочих заданий, оказывается выше. Но, с другой стороны, потеря отдельного узла будет иметь более заметные последствия для доступной мощности вашего кластера.

У мелких узлов также более высокая доля *заблокированных ресурсов* — порций памяти и процессорного времени, которые не используются и которые слишком малы для того, чтобы pod-оболочка могла их занять.

Практика показывает, что узлы лучше делать достаточно большими для выполнения как минимум пяти ваших типичных pod-оболочек, удерживая долю заблокированных ресурсов в пределах 10 %. Если узел способен выполнять десять и более pod-оболочек, объем заблокированных ресурсов будет меньше 5 %.

По умолчанию в Kubernetes действует лимит 110 pod-оболочек на узел. Вы можете поднять его с помощью параметра `--max-pods` утилиты `kubelet`, но в некоторых управляемых сервисах этого сделать нельзя. В целом значения по умолчанию лучше не трогать, если для их изменения нет серьезных причин.

Лимит на количество pod-оболочек для каждого узла означает, что вы можете потерять преимущества от использования самых крупных серверов, которые предлагает ваш облачный провайдер. Вместо этого для лучшей эффективности рассмотрите возможность запустить большее количество узлов помельче. Например, вместо шести узлов с восьмью vCPU можно выбрать 12 узлов с четырьмя vCPU.

Взгляните на относительное потребление ресурсов каждым узлом, используя приборную панель вашего облачного провайдера или команду kubectl top nodes. Чем больше заняты ваши процессоры, тем выше эффективность. Если большие узлы вашего кластера загружены сильнее других, советуем удалить некоторые из узлов помельче и заменить их более крупными.

С другой стороны, если большие узлы имеют низкую степень загруженности, это может означать, что ваш кластер слишком мощный. Поэтому вы можете либо убрать некоторые узлы, либо уменьшить их размер, чтобы сократить общие расходы.

Рекомендуемый подход

Крупные узлы обычно оказываются более экономичными, поскольку на накладные системные расходы у них приходится меньшая доля ресурсов. Подбирайте размеры ваших узлов, анализируя реальную нагрузку на ваш кластер, стремитесь к показателю 10–100 pod-оболочек на узел.

Оптимизация хранилища

Объем дискового хранилища — одна из статей облачных расходов, которую часто игнорируют. Облачные провайдеры предлагают разные объемы дискового пространства для каждого типа серверов, и цена на крупномасштабные хранилища тоже варьируется.

Вы можете достичь довольно высокой загруженности процессора и памяти с помощью запросов и лимитов на ресурсы в Kubernetes, однако с хранилищем все обстоит иначе и многим узлам выделяется слишком много дискового пространства.

Однако дело не только в переизбытке свободного места, но и в классе самого хранилища. Большинство облачных провайдеров предлагают разные классы хранилищ с разной пропускной способностью или числом операций ввода/вывода в секунду (I/O operations per second, IOPS).

Например, базам данных, которые используют постоянные дисковые тома, часто требуется очень высокий показатель IOPS для высокоскоростного доступа к хранилищу. Это дорого. Вы можете сэкономить на облачных расходах, если выделите хранилища с низким уровнем IOPS для рабочих заданий, которые не требуют особой пропускной способности. С другой стороны, если ваше приложение демонстрирует низкую производительность, тратя много времени в ожидании дискового ввода/вывода, вероятно, вам следует выделить больше IOPS.

Консоль вашего облачного провайдера или сервиса Kubernetes обычно умеет показывать количество IOPS, используемое вашими узлами. С помощью этих цифр можно решить, стоит ли экономить.

В идеале у вас должна быть возможность устанавливать запросы ресурсов для контейнеров, которым нужна высокая пропускная способность или большой объем хранилища. Однако на сегодня Kubernetes не позволяет это делать, хотя в будущем может быть добавлена поддержка запросов IOPS.

Рекомендуемый подход

Не используйте типы серверов, размер хранилищ которых превышает ваши потребности. Выделяйте дисковые тома с как можно меньшим показателем IOPS, учитывая пропускную способность и пространство, которые вы действительно используете.

Избавление от неиспользуемых ресурсов

По мере увеличения вашего кластера Kubernetes вы будете обнаруживать множество неиспользуемых или *потерянных* ресурсов, которые разбросаны по темным углам. Если такие ресурсы не подчищать, со временем они составят существенную долю ваших общих расходов.

На самом высоком уровне можно найти облачные серверы, которые не входят ни в один кластер — ведь очень легко забыть об удалении сервера, который больше не используется.

Другие типы облачных ресурсов, такие как балансировщики нагрузки, публичные IP-адреса и дисковые тома, тоже стоят денег, даже если простаивают без дела. Вы должны регулярно просматривать ресурсы всех типов и избавляться от тех из них, которые не используются.

Точно так же в вашем кластере Kubernetes могут быть развертывания и pod-оболочки, которые не получают никакого трафика, так как на них не ссылается ни один сервис.

Даже незапущенные образы контейнеров занимают дисковое пространство на ваших узлах. К счастью, Kubernetes автоматически подчищает неиспользуемые образы при нехватке места на диске[1].

Использование метаданных владельца

Чтобы минимизировать неиспользуемые ресурсы, на уровне организации стоит ввести правило, согласно которому каждый ресурс должен иметь информацию о своем владельце. Для этого можно использовать аннотации Kubernetes (см. подраздел «Метки и аннотации» на с. 209).

Например, каждое развертывание можно было бы пометить так:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: my-brilliant-app
  annotations:
    example.com/owner: "Customer Apps Team"
...
```

[1] Это поведение можно отрегулировать с помощью параметров сборки мусора kubelet (kubernetes.io/docs/concepts/cluster-administration/kubelet-garbage-collection/).

Метаданные владельца должны указывать на человека или команду, с которыми можно связаться при возникновении вопросов относительно конкретного ресурса. Это и само по себе полезно, но особенно удобно, когда нужно выявить заброшенные или неиспользуемые ресурсы (стоит отметить, что к пользовательским аннотациям лучше добавлять префикс в виде доменного имени вашей компании наподобие example.com, чтобы избежать конфликтов с другими аннотациями, которые могут иметь такое же имя).

Вы можете регулярно запрашивать у кластера все ресурсы, у которых нет аннотации владельца, и добавлять их в список на потенциальное удаление. Если строго следовать этому правилу, можно немедленно избавиться от всех «ничейных» ресурсов. Но лучше этого не делать, особенно в самом начале: благосклонность разработчиков — такой же важный ресурс, как и мощность кластера, а может, даже важнее.

Рекомендуемый подход

Установите аннотации владельца для всех своих ресурсов; укажите в них информацию о том, к кому следует обращаться, если с тем или иным ресурсом возникнут проблемы или если он выглядит заброшенным и подлежащим удалению.

Поиск малоиспользуемых ресурсов

Некоторые ресурсы могут получать очень низкий уровень трафика или вообще быть изолированными. Возможно, они были отсоединены от клиентской части сервиса из-за изменений в метках, а может, были временными или экспериментальными.

Каждая pod-оболочка должна предоставлять в виде показателя количество полученных запросов (подробнее — в главе 16). Используйте эти показатели для поиска pod-оболочек, которые получают мало трафика или совсем его не получают, и создайте список ресурсов для потенциального удаления.

Вы также можете проверить показатели загруженности процессора и памяти у каждой pod-оболочки в вашей веб-консоли, чтобы найти наименее загруженные из них: pod-оболочки, которые ничего не делают, — явно нерациональное использование ресурсов.

Если метаданные владельца есть, свяжитесь с ним и уточните, действительно ли нужны те или иные pod-оболочки (они, например, могут предназначаться для приложения, которое еще находится в разработке).

Вы можете применить еще одну пользовательскую аннотацию Kubernetes (например, `example.com/lowtraffic`), чтобы пометить pod-оболочки, которые не получают никаких запросов, но нужны по той или иной причине.

Рекомендуемый подход

Регулярно ищите в своем кластере малозагруженные или заброшенные ресурсы и избавляйтесь от них. В этом могут помочь аннотации владельца.

Удаление завершенных запланированных заданий

Запланированные задания в Kubernetes (см. подраздел «Запланированные задания» на с. 220) — это pod-оболочки, которые выполняются единожды и не перезапускаются после завершения работы. Однако в базе данных Kubernetes остаются `Job`-объекты, и если у вас наберется значительное количество завершенных заданий, это может повлиять на производительность API. Для удаления таких объектов предусмотрен удобный инструмент `kube-job-cleaner` (github.com/hjacobs/kube-job-cleaner).

Проверка резервной мощности

У вашего кластера всегда должно быть достаточно резервной мощности, чтобы справиться с отказом одного рабочего узла. Для проверки остановите свой самый большой узел (см. пункт «Обратное масштабирование» на с. 144). Как только из узла будут выселены все pod-оболочки, убедитесь в том, что ваши приложения по-прежнему находятся в рабочем состоянии с тем количеством реплик, которое указано в конфигурации. Если это не так, вам следует выделить дополнительные мощности для своего кластера.

Если у ваших сервисов нет возможности перераспределить рабочие задания при отказе узла, они в лучшем случае потеряют часть производительности, а в худшем вовсе станут недоступными.

Использование зарезервированных серверов

Некоторые облачные провайдеры предлагают разные классы серверов с разными жизненными циклами. Выбор *зарезервированных* серверов — это разумный компромисс между ценой и гибкостью.

Например, в AWS зарезервированные серверы стоят примерно в два раза дешевле, чем стандартные, которые выделяются *по требованию*. Вы можете резервировать серверы на разные периоды времени: на год, на три года и т. д. В AWS они имеют фиксированный размер, поэтому, если через несколько месяцев окажется, что вам нужны более крупные узлы, ваше резервирование будет по большей части бесполезным.

Аналог зарезервированных серверов от Google Cloud подразумевает *скидки для тех, кто обязуется их использовать*. Вы можете внести предоплату за определенное количество виртуальных процессоров и объем памяти. Это более гибкий подход, чем в AWS, так как он позволяет выходить за рамки зарезервированных ресурсов: просто за все, что не было зарезервировано заранее, вы по запросу платите обычную цену.

Зарезервированные серверы и скидки за обязательное использование могут быть хорошим выбором, если вы знаете свои требования на ближайшее будущее. Однако расходы на резервирование, которое в итоге вам не пригодилось, не возвращаются, а за весь период необходимо платить авансом. Поэтому серверы следует резервировать только на то время, на протяжении которого ваши требования, скорее всего, не претерпят существенных изменений.

Но если вы можете планировать на год или два вперед, использование этого типа серверов может заметно сократить ваши расходы.

Рекомендуемый подход

Используйте зарезервированные серверы, если вы определились с вашими потребностями на ближайшие год-два. Но имейте в виду, что впоследствии вы не сможете что-то изменить или получить свои деньги назад.

Использование прерываемых (spot) серверов

Spot-серверы, по терминологии AWS, или *прерываемые ВМ*, как их называет Google, не дают никаких гарантий доступности и часто имеют ограниченное время жизни. Таким образом, их использование является компромиссом между ценой и доступностью.

Spot-серверы дешевые, но их работа может быть приостановлена или возобновлена в любой момент, к тому же они могут быть и вовсе удалены. К счастью, платформа Kubernetes спроектирована так, чтобы обеспечивать высокую доступность, несмотря на потерю отдельных узлов кластера.

Колебание цены и частоты прерываний

Таким образом, spot-серверы могут стать экономным выбором для вашего кластера. В AWS почасовая тарификация колеблется в зависимости от спроса: если спрос повышается на определенный вид серверов в конкретном регионе и зоне доступности, цена растет.

Прерываемые ВМ в Google Cloud, в свою очередь, тарифицируются по фиксированной ставке, но частота прерываний может меняться: Google говорит, что в среднем за неделю будет выгружено около 5–15 % ваших узлов (cloud.google.com/compute/docs/instances/preemptible). Но в то же время прерываемые ВМ могут быть на 80 % дешевле серверов, которые выделяются по требованию (зависит от типа сервера).

Прерываемые узлы могут сократить ваши расходы вдвое

Использование прерываемых узлов в кластере Kubernetes может оказаться крайне эффективным с точки зрения экономии денежных средств. Возможно, вам придется запускать дополнительные узлы для того, чтобы рабочие задания могли пережить преждевременную остановку, но, как показывает практика, при этом можно достичь 50%-ного сокращения расходов на каждом из узлов.

Вы также можете обнаружить, что прерываемые узлы хорошо подходят для интеграции небольшой порции хаотического проектирования в ваш кластер (см. раздел «Хаотическое тестирование» на с. 152) — при условии, что ваше приложение заранее к этому подготовлено.

Но имейте в виду, что у вас всегда должно быть достаточно непрерываемых узлов для того, чтобы справиться с минимальным объемом рабочей нагрузки. Никогда не ставьте на кон то, потерю чего не можете себе позволить. Если у вас есть много прерываемых серверов, хорошей идеей будет использовать автомасштабирование кластера, чтобы любой выгруженный узел был как можно быстрее заменен (см. пункт «Автомасштабирование» на с. 145).

Теоретически *все* прерываемые узлы могут исчезнуть одновременно. Поэтому, несмотря на экономию, лучше ограничить количество вытесняемых узлов — например, двумя третями вашего кластера.

Рекомендуемый подход

Снижайте расходы за счет использования прерываемых или spot-серверов для некоторых из своих узлов — но только в тех объемах, потеря которых является допустимой. Всегда имейте в своем кластере и непрерываемые узлы.

Использование принадлежности к узлам для управления планированием

В Kubernetes можно использовать концепцию *принадлежности узлов* (node affinities), чтобы pod-оболочки, отказ которых недопустим, не размещались на прерываемых узлах (см. раздел «Принадлежность к узлам» на с. 210).

Например, в Google Kubernetes Engine прерываемые серверы помечены как `cloud.google.com/gke-preemptible`. Чтобы планировщик Kubernetes никогда не разместил pod-оболочку на одном из таких узлов, добавьте следующий фрагмент в ее спецификацию (или в спецификацию развертывания):

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: cloud.google.com/gke-preemptible
          operator: DoesNotExist
```

Принадлежность `requiredDuringScheduling...` является обязательной: pod-оболочка, в которой она указана, никогда не будет запущена на узле, соответствующем выражению селектора (это называется *жесткой принадлежностью*).

Как вариант, Kubernetes можно сообщить о том, что некоторые из ваших менее важных pod-оболочек, то есть те, чью работу время от времени допустимо останавливать, могут быть размещены на прерываемых узлах. На этот случай предусмотрена *мягкая принадлежность* с противоположным смыслом:

```
affinity:
  nodeAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - preference:
        matchExpressions:
        - key: cloud.google.com/gke-preemptible
          operator: Exists
      weight: 100
```

По существу, это означает следующее: «Если можешь — пожалуйста, размещай данную pod-оболочку на прерываемом узле; если нет — ничего страшного».

Рекомендуемый подход

Если у вас есть прерываемые узлы, используйте концепцию принадлежности в Kubernetes, чтобы ваши критически важные рабочие задания не прерывались.

Балансировка вашей рабочей нагрузки

Мы уже говорили о том, что задача планировщика в Kubernetes — сделать так, чтобы нагрузка распределялась равномерно между как можно большим количеством узлов и чтобы копии одной и той же pod-оболочки размещались на разных узлах (для высокой доступности).

В целом планировщик отлично с этим справляется, но бывают особенные случаи, которые требуют отдельного внимания.

Предположим, что у вас есть два узла и два сервиса, А и Б, по две реплики у каждого. В сбалансированном кластере на каждом узле будет размещаться по одной реплике сервиса А и сервиса Б (рис. 5.1). Если один узел откажет, А и Б останутся доступными.

Пока все хорошо. Но представьте, что у вас отказал второй узел. Планировщик заметит, что сервисам А и Б нужны дополнительные реплики и их можно разместить лишь на одном узле, что он и сделает. Теперь первый узел выполняет по две реплики сервисов А и Б.

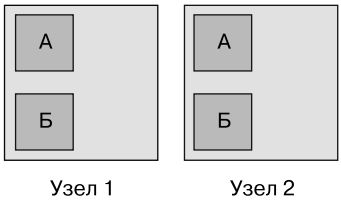

Рис. 5.1. Сервисы А и Б распределены между доступными узлами

Теперь предположим, что мы запустили новый узел на замену второму, который вышел из строя. Даже когда он станет доступным, на нем не будет ни одной pod-оболочки: планировщик никогда не перемещает работающие pod-оболочки с одного узла на другой.

У нас получился разбалансированный кластер (itnext.io/keep-you-kubernetes-cluster-balanced-the-secret-to-high-availability-17edf60d9cb7), первый узел которого выполняет все pod-оболочки, а второй пустует (рис. 5.2).

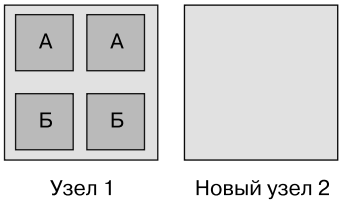

Рис. 5.2. После отказа второго узла все реплики переходят на первый

Но на этом наши беды не заканчиваются. Представьте, что вы развертываете плавающее обновление для сервиса А (назовем его новую версию А*). Планировщику нужно запустить две обновленные реплики А*, подождать, пока они будут готовы, и удалить их старые версии. Новые версии попадут на второй узел, поскольку он бездействует, тогда как узел 1 уже выполняет четыре pod-оболочки. Итак, две новые реплики

сервиса А* стартуют на узле 2, а их предшественницы удаляются из узла 1 (рис. 5.3).

Вы оказались в плохой ситуации, поскольку каждый сервис размещает все свои реплики на одном узле (А* на узле 2, а Б — на узле 1). Несмотря на наличие двух узлов, вы теряете высокую доступность, ведь отказ любого из узлов приведет к перебоям в работе одного из сервисов.

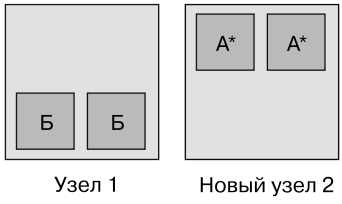

Рис. 5.3. После выкатывания сервиса А* кластер все еще разбалансирован

Суть проблемы в том, что планировщик перемещает pod-оболочки с одного узла на другой, только если они по какой-либо причине перезапускаются. К тому же задача планировщика — распределить рабочую нагрузку равномерно между узлами — иногда конфликтует с поддержанием высокой доступности отдельных сервисов.

Одним из решений является использование инструмента под названием Descheduler (github.com/kubernetes-sigs/descheduler). Вы можете запускать его время от времени в виде запланированных заданий Kubernetes, и он постарается перебалансировать кластер, найдя pod-оболочки, которые необходимо переместить и удалить.

Для Descheduler можно задавать различные стратегии и линии поведения. Например, одна из его политик ищет малонагруженные узлы и переносит на них pod-оболочки с других узлов (pod-оболочки при этом удаляются).

Другая политика ищет pod-оболочки, две и более реплики которых размещены на одном узле, и выселяет их. Это решает проблему, проиллюстрированную в нашем примере, когда рабочие задания формально сбалансированы, но на самом деле ни один из сервисов не является высокодоступным.

Резюме

Kubernetes довольно успешно справляется с вашими рабочими нагрузками надежным и эффективным способом, не требующим ручного вмешательства. Если предоставить планировщику Kubernetes точную оценку потребностей ваших контейнеров, в большинстве случаев он сам сделает все необходимое.

Таким образом, время, которое у вас ушло бы на решение инфраструктурных проблем, можно потратить с большей пользой — например, на разработку приложений. Спасибо, Kubernetes!

Понимание того, как Kubernetes управляет ресурсами, является ключом к правильному построению и выполнению кластера. Вот самое важное, что следует вынести из этой главы.

- Kubernetes выделяет контейнерам ресурсы процессора и памяти на основе *запросов* и *лимитов*.
- Запросы контейнера — это минимальный объем ресурсов, необходимый для его работы. Лимиты — максимальный объем, который ему позволено использовать.
- Минимальный образ контейнера быстрее собирается, загружается, развертывается и запускается. Чем меньше контейнер, тем меньше в нем потенциальных уязвимостей безопасности.
- Проверки работоспособности говорят Kubernetes о том, насколько корректно работает контейнер. Если такая проверка завершается неудачно, контейнер удаляется и запускается заново.
- Проверки готовности говорят Kubernetes о том, способен ли контейнер обслуживать запросы. Если такая проверка завершается неудачно, контейнер извлекается из сервисов, которые на него ссылаются, и таким образом ограждается от пользовательского трафика.
- `PodDisruptionBudget` позволяет ограничить количество pod-оболочек, которые можно останавливать одновременно в процессе *выселения*. Это позволяет поддерживать высокую доступность вашего приложения.
- Пространства имен — это механизм логического разделения кластера. Их можно создавать для каждого приложения или группы связанных между собой приложений.
- Для обращения к сервису из другого пространства имен можно использовать доменное имя — например, `SERVICE.NAMESPACE`.
- `ResourceQuota` позволяет задавать общие лимиты на ресурсы для отдельного пространства имен.
- `LimitRange` определяет запросы и лимиты на ресурсы по умолчанию, относящиеся к контейнеру в пространстве имен.
- Устанавливайте лимиты на ресурсы так, чтобы в ходе нормальной работы ваше приложение почти (но не до конца) их достигало.
- Не выделяйте места в облаке больше, чем вам нужно, и не выбирайте высокопроизводительное хранилище, если оно не требуется для работы вашего приложения.
- Устанавливайте аннотации владельца для всех своих ресурсов. Ищите бесхозные ресурсы, регулярно сканируя свой кластер.

- Найдите и удалите ресурсы, которые не используются (но сначала уточните информацию у их владельцев).
- Зарезервированные серверы могут сэкономить вам деньги, если вы планируете использование ресурсов в долгосрочной перспективе.
- Прерываемые серверы, вероятно, сразу снизят ваши денежные расходы, но будьте готовы к тому, что такие серверы в ближайшее время могут исчезнуть. Используйте концепцию принадлежности, чтобы не размещать отказочувствительные pod-оболочки на прерываемых узлах.

6 Работа с кластерами

> Если «Тетрис» меня чему-то и научил, так это тому, что ошибки накапливаются, а достижения исчезают.
>
> *Эндрю Клей Шафер*

Как понять, хорошо ли работает ваш кластер Kubernetes и все ли с ним в порядке? Как его масштабировать в зависимости от нагрузки, но одновременно и минимизировать облачные расходы? В этой главе мы рассмотрим проблемы, связанные с администрированием кластеров Kubernetes в промышленных условиях, и некоторые из инструментов, которые могут помочь в данном деле.

Как мы уже видели в главе 3, при работе с кластером Kubernetes следует принимать во внимание много важных моментов: доступность, аутентификацию, обновления и т. д. Если вы используете этот хорошо управляемый сервис, следуя нашим рекомендациям, большинство проблем уже были решены за вас.

Но что делать с кластером, решать именно вам. В этой главе вы научитесь масштабировать и менять размеры своего кластера, проверять его на соответствие, искать угрозы безопасности и тестировать устойчивость своей инфраструктуры с помощью Chaos Monkey.

Масштабирование и изменение размеров кластера

Насколько большим должен быть ваш кластер? При самостоятельном размещении Kubernetes и использовании почти любых управляемых сервисов текущие расходы кластера напрямую зависят от количества и размера его узлов. Если мощность кластера слишком низкая, ваши рабочие задания не смогут нормально выполняться

или будут отказывать под тяжелой нагрузкой. Если мощность слишком высокая, вы станете терять деньги.

Разумное масштабирование и изменение размеров кластера является очень важным процессом. Давайте посмотрим, какие решения в него вовлечены.

Планирование мощности

Чтобы дать начальную оценку необходимой вам мощности, подумайте о том, сколько традиционных серверов потребуется для выполнения одних и тех же приложений. Например, если ваша текущая архитектура состоит из десяти облачных серверов, кластеру Kubernetes, вероятно, не понадобится больше десяти узлов для обслуживания той же нагрузки; плюс еще один в качестве резерва. На самом деле это, скорее всего, перебор: благодаря равномерному распределению нагрузки между узлами Kubernetes может достичь более высокой загруженности, чем традиционные серверы. Но чтобы откалибровать кластер для оптимальной мощности, вам могут понадобиться некоторое время и опыт.

Минимальный кластер

Свой первый кластер вы, вероятно, будете использовать для упражнений и экспериментов, а также для того, чтобы понять, как запустить в нем собственное приложение. Поэтому не стоит выбрасывать деньги на крупные кластеры до тех пор, пока не получите какое-то представление о необходимой вам мощности.

Минимально возможный кластер Kubernetes состоит из одного узла. Он позволяет опробовать эту платформу и запустить на ней небольшие задания для разработки, как мы видели в главе 2. Однако одноузловой кластер не обладает устойчивостью к отказу оборудования самого узла, сервера Kubernetes API или `kubelet` (агента-демона, отвечающего за запуск рабочих заданий на каждом узле).

Если вы пользуетесь сервисами по управлению Kubernetes, такими как GKE (см. подраздел «Google Kubernetes Engine (GKE)» на с. 74), вам не нужно волноваться о выделении ведущих узлов, это делается за вас. Но если вы занимаетесь построением собственного кластера, вам следует решить, сколько ведущих узлов у него будет.

Для поддержания устойчивости кластеру Kubernetes требуется минимум три узла: один был бы неустойчив, два — не сошлись бы на том, кто должен быть лидером; поэтому нужно три узла.

Хотя это и не рекомендуется, но даже на таком мелком кластере можно выполнять полезную работу. Но все же лучше добавить несколько рабочих узлов, чтобы

ваши собственные задания не конкурировали за ресурсы с управляющим уровнем Kubernetes.

При условии, что ваш управляющий уровень является высокодоступным, вы *можете* ограничиться одним рабочим узлом. Однако, чтобы защититься от поломки сервера и позволить Kubernetes выполнять хотя бы по две реплики для каждой pod-оболочки, этот показатель разумнее довести хотя бы до двух. Чем больше узлов, тем лучше, особенно учитывая тот факт, что планировщик Kubernetes не всегда может гарантировать полноценный баланс нагрузки между всеми доступными узлами (см. подраздел «Балансировка вашей рабочей нагрузки» на с. 133).

Рекомендуемый подход

Кластерам Kubernetes нужно как минимум три ведущих узла для достижения высокой доступности; а для того, чтобы справиться с работой крупных кластеров, может понадобиться еще больше. Чтобы сделать ваши рабочие задания устойчивыми к сбоям отдельных серверов, необходимы по меньшей мере два рабочих узла, а лучше три.

Максимальный кластер

Есть ли какое-то ограничение относительно размеров кластера Kubernetes? Если коротко, то да, но вам почти наверняка не стоит об этом волноваться; Kubernetes версии 1.12 официально поддерживает кластеры до 5000 узлов. Поскольку кластеризация требует взаимодействия между узлами, вместе с кластером экспоненциально растут количество возможных коммуникационных маршрутов и совокупная нагрузка на внутреннюю базу данных. Кластер Kubernetes, вероятно, *будет* функционировать и с большим, чем 5000, количеством узлов, но в условиях промышленных нагрузок его работа или по крайней мере достаточная отзывчивость не *гарантируется*.

В документации к Kubernetes говорится о том, что поддерживаемая конфигурация кластера (kubernetes.io/docs/setup/best-practices/cluster-large) не должна превышать 5000 узлов, 150 000 pod-оболочек, 300 000 контейнеров, а на одном узле не должно быть больше 100 pod-оболочек. Стоит помнить о том, что чем больше кластер, тем выше нагрузка на его ведущие узлы: если вы сами занимаетесь ведущими узлами, их следует размещать на довольно мощных компьютерах, чтобы они могли справиться с кластерами из тысячи узлов.

Рекомендуемый подход

Для максимальной надежности ваши кластеры не должны иметь более 5000 узлов и 150 000 pod-оболочек (для большинства пользователей это не проблема). Если нужны еще ресурсы, создайте несколько кластеров.

Федеративные кластеры

Если ваши рабочие задания имеют крайне высокие требования или вам надо работать в огромных масштабах, подобные ограничения могут стать реальной проблемой. В таком случае стоит создать несколько кластеров Kubernetes и объединить их в *федерацию*, чтобы между ними можно было реплицировать рабочую нагрузку.

Федерация позволяет синхронизировать два и более кластера, запуская на них идентичные задания. Это может пригодиться, если вам нужны кластеры Kubernetes в разных облаках (для устойчивости) или в разных географических зонах (чтобы снизить время отклика для ваших пользователей). Федерация может продолжать работу, даже если откажет ее отдельный кластер.

Больше о федерации кластеров в Kubernetes можно почитать в документации (kubernetes.io/docs/concepts/cluster-administration/federation).

Для большинства пользователей Kubernetes федерация не является насущной необходимостью. Когда дело доходит до очень крупных масштабов, почти все могут справиться со своими рабочими заданиями с помощью нескольких независимых кластеров по несколько сотен или тысяч узлов в каждом.

Рекомендуемый подход

Если вам нужно реплицировать рабочую нагрузку между несколькими кластерами (например, для георезервирования или чтобы снизить время отклика), используйте федерацию. Но большинству пользователей эта возможность не нужна.

Нужно ли мне больше одного кластера?

Если вы не работаете в очень крупных масштабах, о чем мы упоминали в предыдущем разделе, у вас, скорее всего, нет необходимости иметь больше одного или двух кластеров: возможно, один кластер понадобится для промышленной среды, а другой — для финальной проверки и тестирования.

Для удобства и простоты управления ресурсами кластер можно разделить на логические части, которые называются *пространствами имен* (подробно о них рассказывается в разделе «Использование пространств имен» на с. 117). За редким исключением, накладные расходы на администрирование нескольких кластеров себя не оправдывают.

Существуют сценарии, такие как соблюдение требований безопасности и правовых норм, когда следует позаботиться о том, чтобы сервисы одного кластера были полностью изолированы от сервисов другого (например, при работе с конфиденциальными медицинскими записями или при запрете передачи данных из одной географической зоны в другую по юридическим соображениям). В таких ситуациях необходимо создавать отдельные кластеры. Большинство пользователей Kubernetes не сталкиваются с подобными проблемами.

Рекомендуемый подход

Используйте по одному кластеру для промышленной среды и финальных проверок, если только вам действительно не нужно полностью изолировать одну группу рабочих заданий или разработчиков от другой. Если же вы хотите разделить свой кластер, чтобы им было легче управлять, используйте пространства имен.

Узлы и серверы

Чем большей мощностью обладает узел, тем больше работы он может выполнить. Мощность выражается в количестве процессорных ядер (виртуальных или физических), объеме доступной памяти и в меньшей степени — в количестве дискового пространства. Но что лучше — десять очень больших узлов или, к примеру, сто гораздо меньших?

Выбор узлов подходящего размера

Не существует такого универсального размера узлов, который бы подходил для всех кластеров Kubernetes: все зависит от вашего облачного провайдера или поставщика оборудования, а также от конкретных рабочих нагрузок.

Стоимость разных типов серверов (с поправкой на мощность) способна повлиять на ваше решение относительно размера узлов. Например, какие-то облачные провайдеры могут предложить небольшую скидку для более крупных серверов и те в результате окажутся дешевле по сравнению с большим количеством мелких узлов — в случае, если ваши рабочие задания очень ресурсоемкие.

То, сколько узлов должно быть в кластере, также влияет на выбор их размера. Чтобы воспользоваться преимуществами Kubernetes, такими как репликация pod-оболочек и высокая доступность, вы должны распределять нагрузку между несколькими узлами. Но узлы имеют слишком много незадействованных ресурсов, и это будет пустой тратой денег.

Если для высокой доступности вам нужно, скажем, десять узлов, но на каждом из них будет работать лишь несколько pod-оболочек, узлы можно сделать очень маленькими. С другой стороны, если вам нужно лишь два узла, вы можете сделать их довольно крупными и тем самым потенциально сэкономить на более выгодных тарифах для серверов.

Рекомендуемый подход

Используйте экономные типы серверов, которые предлагает ваш провайдер. Часто более крупные узлы оказываются дешевле, но если их у вас немного, добавьте к ним несколько узлов поменьше, чтобы улучшить избыточность.

Типы облачных серверов

Поскольку компоненты Kubernetes, такие как kubelet, сами по себе используют определенное количество ресурсов, да и для выполнения полезной работы вам понадобится дополнительная мощность, скорее всего, самый мелкий тип серверов, предлагаемый облачным провайдером, вам не подойдет.

Ведущие узлы в небольших кластерах (до пяти узлов) должны иметь как минимум один виртуальный процессор (vCPU) и 3–4 ГиБ памяти. В более крупных кластерах каждому ведущему узлу нужно больше памяти и процессоров. Это эквивалентно серверу `n1-standard-1` в Google Cloud, `m3.medium` в AWS и Standard DS1 v2 в Azure.

Однопроцессорный сервер с 4 ГиБ памяти является разумным минимумом и для рабочего узла. Хотя, как мы уже говорили, выделение узлов покрупнее может оказаться более выгодным с финансовой точки зрения. В Google Kubernetes Engine, например, по умолчанию используются серверы `n1-standard-1`, которые примерно соответствуют этим спецификациям.

Для более крупных кластеров, например с несколькими десятками узлов, может иметь смысл использовать сочетание из серверов 2–3 типов. Это означает, что pod-оболочки с более ресурсоемкими рабочими заданиями, требующими большего количества памяти, платформа Kubernetes может размещать на крупных узлах, тогда как узлы помельче будут доступны для менее требовательных pod-оболочек (см. раздел «Принадлежность к узлам» на с. 210). Это дает планировщику Kubernetes максимальную свободу выбора при распределении той или иной pod-оболочки.

Гетерогенные узлы

Не все узлы одинаково полезны. Вам могут понадобиться серверы с особыми характеристиками, такие как графический адаптер (GPU). Графические адаптеры являются высокопроизводительными параллельными процессорами, которые широко используются для ресурсоемких задач вроде машинного обучения или анализа данных, не имеющих ничего общего с графикой.

С помощью *лимитов на ресурсы*, встроенных в Kubernetes (см. подраздел «Лимиты на ресурсы» на с. 109), можно указать, что определенной pod-оболочке нужен, к примеру, как минимум один графический адаптер. Благодаря этому такие pod-оболочки будут размещаться только на узлах с GPU, получая повышенный приоритет по сравнению с оболочками, способными работать на любом сервере.

Основная часть узлов Kubernetes, скорее всего, работает под управлением той или иной версии Linux, что подходит для большинства приложений. Но, как вы помните, контейнеры — это *не* виртуальные машины, поэтому находящиеся внутри них процессы выполняются непосредственно на ядре операционной системы сервера. Например, исполняемые файлы Windows не запустятся на узле с Linux, поэтому, если вам нужны Windows-контейнеры, придется выделить для них узлы под управлением Windows.

Рекомендуемый подход

Большинство контейнеров рассчитаны на Linux, поэтому вы, вероятно, захотите использовать узлы на основе этой системы. Может потребоваться добавить один или несколько специальных типов серверов с особыми характеристиками, например с наличием GPU или Windows.

Физические серверы

Одно из самых полезных свойств Kubernetes состоит в возможности соединять компьютеры разных размеров, архитектур и мощности, предоставляя единый унифицированный вычислительный модуль для выполнения рабочих заданий. Kubernetes обычно ассоциируется с облачными серверами, однако многие организации размещают в вычислительных центрах реальные, «железные» компьютеры, которые потенциально можно использовать в кластерах Kubernetes.

В главе 1 мы видели, как организации благодаря технологиям постепенно отходят от покупки компьютеров и все чаще начинают арендовать вычислительные

мощности. И это имеет смысл с финансовой точки зрения. Однако, если ваша компания уже владеет большим количеством физических серверов, вам пока не стоит списывать их со счетов: вместо этого подумайте о том, чтобы присоединить их к кластеру Kubernetes (см. подраздел «Локально размещенные физические серверы» на с. 84).

Рекомендуемый подход

Если у вас имеются аппаратные серверы с излишком ресурсов или если вы еще не готовы полностью мигрировать в облако, используйте Kubernetes для выполнения контейнерных рабочих заданий на ваших собственных компьютерах.

Масштабирование кластера

Итак, вы подобрали разумный начальный размер для своего кластера и правильное сочетание типов серверов для своих рабочих узлов. Можно ли на этом останавливаться? Почти наверняка нет: вам, вероятно, придется сокращать или расширять свой кластер соответственно изменениям нагрузки или бизнес-требований.

Группы серверов

Добавлять узлы в кластер Kubernetes довольно легко. Если вы размещаете Kubernetes самостоятельно, инструменты для управления, такие как kops (см. подраздел «kops» на с. 78), могут делать это за вас. У kops есть концепция *группы серверов* — набора узлов определенного типа (например, `m3.medium`). Управляемые сервисы, такие как Google Kubernetes Engine, предлагают аналогичный механизм под названием *«пулы узлов»*.

Вы можете масштабировать группы серверов или пулы узлов, изменяя их минимальный и максимальный размеры и/или выбирая другие типы конфигураций.

Обратное масштабирование

В принципе, у Kubernetes нет никаких проблем и с обратным масштабированием. Вы можете приказать системе *очистить* узлы, подлежащие удалению, и она постепенно выключит pod-оболочки на этих узлах или куда-нибудь их переместит.

Большинство инструментов для управления кластером выполняют очищение узлов автоматически, но также можно использовать команду `kubectl drain`, чтобы сделать это вручную. Если у кластера достаточно свободной мощности для пере-

распределения «обреченных» pod-оболочек, вы сможете удалить их сразу после успешного очищения узлов.

Чтобы избежать чрезмерного сокращения количества реплик pod-узлов для заданного сервиса, можно использовать ресурс `PodDisruptionBudget`: укажите минимальное число доступных pod-оболочек или разрешенное количество *недоступных* pod-оболочек в любой момент времени (см. подраздел «Ресурс PodDisruptionBudget» на с. 115).

Если Kubernetes в результате очищения узла превысит эти лимиты, операция очистки заблокируется до их изменения или пока в кластере не освободится больше ресурсов.

Очистка позволяет pod-оболочкам корректно завершить свою работу, убрать за собой, а также, если необходимо, сохранить какое-либо состояние. Для большинства приложений это более предпочтительно, чем простое выключение узла, из-за чего pod-оболочки прекращают работу немедленно.

Рекомендуемый подход

Не следует выключать те узлы, которые вам больше не нужны. Сначала очистите их, чтобы рабочие задания могли мигрировать на другие узлы, и убедитесь в том, что в вашем кластере остается достаточно свободных ресурсов.

Автомасштабирование

Большинство облачных провайдеров поддерживают *автомасштабирование* — автоматическое увеличение или уменьшение количества серверов в группе на основе какого-то показателя или графика. Например, группы автомасштабирования AWS (AWS autoscaling groups, ASGs) могут поддерживать минимальное и максимальное число серверов: если один откажет, другой запустится и займет его место, а если серверов станет слишком много, некоторые из них будут выключены.

В качестве альтернативы, если ваша нагрузка колеблется в зависимости от времени суток, можно запланировать расширение и сокращение группы в указанное время: например сконфигурировать масштабирование таким образом, чтобы количество серверов менялось скачкообразно в зависимости от нагрузки. Допустим, если на протяжении 15 минут процессор остается занятым на более чем 90 %, группа может автоматически увеличиваться, пока использование процессора не упадет ниже этого порога. Когда нагрузка снизится, группа может сократиться, чтобы сэкономить ваши деньги.

У Kubernetes есть дополнение Cluster Autoscaler, с помощью которого такие инструменты для управления кластером, как kops, могут выполнять облачное автомасштабирование. Последнее также доступно в управляемых кластерах, таких как Azure Kubernetes Service.

Однако для правильной настройки потребуются время и некоторое число попыток, в то время как для многих пользователей автомасштабирование может оказаться и вовсе ненужным. Большинство кластеров Kubernetes начинают с малого и затем постепенно и плавно добавляют узлы тут и там по мере роста нагрузки.

Хотя для крупных пользователей или приложений с сильно колеблющимися нагрузками автомасштабирование кластера является очень полезной функцией.

Рекомендуемый подход

Не включайте автомасштабирование кластера просто потому, что оно вам доступно: если ваши рабочие нагрузки не слишком колеблются, скорее всего, вы можете обойтись без него. Масштабируйте свой кластер вручную — хотя бы на первых порах, чтобы получить представление о том, насколько со временем изменяются ваши требования к масштабированию.

Проверка на соответствие

Иногда от Kubernetes остается только название. Благодаря своей гибкости система позволяет настраивать кластеры множеством разных способов, и это может создавать проблемы. Проект Kubernetes задумывался как универсальная платформа, поэтому у вас должна быть возможность взять рабочее задание и выполнить его на любом кластере Kubernetes с предсказуемыми результатами. Это означает, что API-вызовы и объекты Kubernetes должны быть доступны и работать одинаково везде.

К счастью, сама платформа Kubernetes содержит набор тестов, позволяющий подтвердить, что кластер *соответствует* спецификации, то есть удовлетворяет основному набору требований для заданной версии Kubernetes. Такие проверки на соответствие крайне полезны для системных администраторов.

Если кластер их не проходит, значит, у вашей конфигурации есть проблема, с которой нужно что-то делать. Когда проверка прошла успешно, вы можете быть уверены в том, что приложения, предназначенные для Kubernetes, будут работать на вашем кластере, а то, что вы написали для своего кластера, заработает и в других местах.

Сертификация CNCF

Cloud Native Computing Foundation (CNCF) является официальным владельцем проекта и торговой марки Kubernetes (см. раздел «Облачная ориентированность» на с. 44), предоставляя различные виды сертификации для продуктов, инженеров и поставщиков, связанных с этой платформой.

Certified Kubernetes

Если вы используете управляемый или частично управляемый сервис Kubernetes, проверьте у него наличие знака качества и логотипа Certified Kubernetes (рис. 6.1). Это изображение говорит о том, что поставщик и сервис отвечают стандарту Certified Kubernetes (github.com/cncf/k8s-conformance), описанному организацией Cloud Native Computing Foundation (CNCF).

Рис. 6.1. Знак качества Certified Kubernetes говорит о том, что продукт или сервис одобрены фондом CNCF

Если продукт содержит в своем названии *Kubernetes*, он должен быть сертифицирован фондом CNCF. Тогда клиенты точно знают, что именно они получают, и уверены в совместимости данного продукта с другими сертифицированными сервисами Kubernetes. Поставщики могут самостоятельно сертифицировать свои продукты с помощью Sonobuoy — инструмента для проверки на соответствие (см. подраздел «Проверка на соответствие с помощью Sonobuoy» на с. 148).

Кроме того, сертифицированные продукты должны отслеживать последнюю версию Kubernetes и предоставлять обновления как минимум раз в год. Знак качества Certified Kubernetes может быть присвоен не только управляемым сервисам, но также инструментам для дистрибуции и установки.

Certified Kubernetes Administrator (CKA)

Чтобы стать сертифицированным администратором Kubernetes, вам необходимо продемонстрировать наличие навыков по управлению кластерами Kubernetes в промышленных условиях, включая установку и конфигурацию, работу с сетью, обслуживание, знание API, безопасность и отладку. Каждый может сдать экзамен CKA, который проводится по Интернету и состоит из набора сложных практических тестов.

CKA имеет репутацию жесткого комплексного экзамена, по-настоящему проверяющего умения и знания. Можете не сомневаться, что любой CKA-сертифицированный инженер действительно знает Kubernetes. Если в вашем бизнесе используется эта платформа, рассмотрите для своих сотрудников (особенно тех, кто непосредственно отвечает за управление кластерами) возможность прохождения программы CKA.

Kubernetes Certified Service Provider (KCSP)

Поставщики могут сами подавать заявки на прохождение программы сертификации Kubernetes Certified Service Provider (KCSP). Для этого они должны быть членами CNCF, предоставлять поддержку от предприятия (например, направляя своих инженеров к месту расположения клиента), активно участвовать в сообществе Kubernetes и иметь в штате не менее трех CKA-сертифицированных инженеров.

Рекомендуемый подход

Ищите знак качества Certified Kubernetes, чтобы быть уверенными в том, что рассматриваемый продукт соответствует стандартам CNCF. Ищите KCSP-сертифицированных поставщиков. Если вы нанимаете администраторов для работы с Kubernetes, обращайте внимание на квалификацию CKA.

Проверка на соответствие с помощью Sonobuoy

Если вы управляете собственным кластером или используете управляемый сервис, но при этом хотите удостовериться в его правильной конфигурации и актуальности версии, можете воспользоваться проверками на соответствие Kubernetes. Стандартным инструментом для выполнения этих проверок является система Sonobuoy от Heptio (github.com/vmware-tanzu/sonobuoy). Чтобы просмотреть результаты в веб-браузере, можно воспользоваться веб-интерфейсом Sonobuoy Scanner (scanner.heptio.com) (рис. 6.2).

Этот инструмент предоставляет для вашего кластера команду `kubectl apply`: с ее помощью выполняются проверки на соответствие. Результаты возвращаются в Heptio, где выводятся в веб-интерфейсе Scanner.

Вероятность того, что какая-то из проверок не будет пройдена, крайне низкая, особенно если вы используете сертифицированный сервис Kubernetes. Но если

подобное все же произойдет, в журнальных записях можно будет найти полезную информацию о том, как исправить проблему.

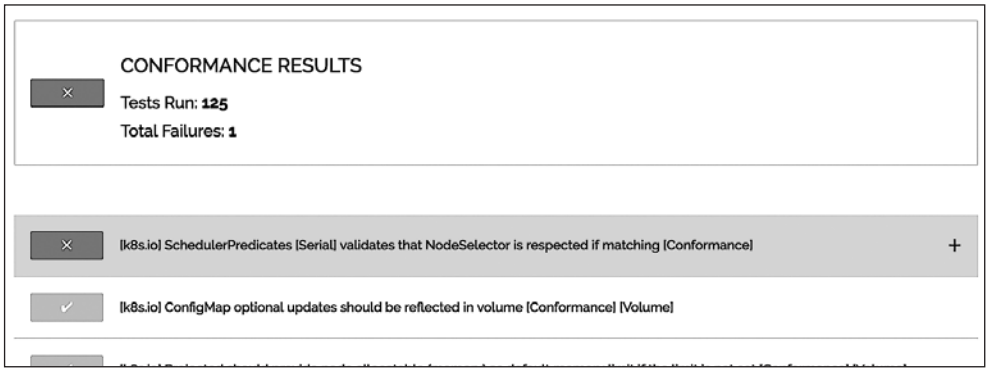

Рис. 6.2. Результаты проверки на соответствие от Sonobuoy Scanner

Рекомендуемый подход

Запустите Sonobuoy после начальной настройки вашего кластера, чтобы убедиться в том, что он соответствует стандартам и все работает. Позже его можно запускать время от времени для уверенности в отсутствии проблем с совместимостью.

Проверка конфигурации и аудит

Соответствие стандартам — это лишь начальное условие, которое должно выполняться любым промышленным кластером. Однако в Kubernetes существует много распространенных проблем с конфигурацией и рабочей нагрузкой, которые не обнаруживаются с помощью проверок на соответствие. Например:

- использование слишком больших образов контейнеров может привести к потере значительного количества времени и ресурсов кластера;
- развертывания с единственной репликой pod-оболочки не являются высокодоступными;
- выполнение процессов в контейнерах от имени администратора представляет потенциальный риск для безопасности (см. раздел «Безопасность контейнеров» на с. 192).

В данном разделе мы рассмотрим некоторые инструменты и методики, с помощью которых вы сможете находить проблемы в вашем кластере, и расскажем, чем эти проблемы вызываются.

K8Guard

Утилита K8Guard, разработанная компанией Target, умеет искать распространенные проблемы в кластерах Kubernetes и либо исправлять их самостоятельно, либо уведомлять о них. Вы можете сконфигурировать ее с учетом конкретных политик вашего кластера (например, можете настроить уведомления на случай, если какой-либо образ контейнера превышает 1 ГиБ или если правило условного доступа позволяет обращаться к ресурсу откуда угодно).

K8Guard также экспортирует показатели, которые можно собирать с помощью такой системы мониторинга, как Prometheus (больше об этом — в главе 16). Благодаря чему, например, можно узнать, сколько развертываний нарушают ваши политики, и оценить производительность ответов Kubernetes API. Это помогает находить и исправлять проблемы на ранних этапах.

Лучше всего сделать так, чтобы утилита K8Guard постоянно работала с вашим кластером и могла предупреждать вас о любых нарушениях, как только они произойдут.

Copper

Copper (copper.sh) — это инструмент для проверки манифестов Kubernetes перед их развертыванием; он отмечает распространенные проблемы или применяет отдельные политики. Copper включает в себя проблемно-ориентированный язык (domain-specific language, DSL), предназначенный для описания правил и политик проверки.

Например, ниже показано правило, написанное на языке Copper, которое блокирует любой контейнер с тегом `latest` (в подразделе «Тег latest» на с. 189 рассказывается, почему это плохая идея):

```
rule NoLatest ensure {
    fetch("$.spec.template.spec.containers..image")
        .as(:image)
        .pick(:tag)
        .contains("latest") == false
}
```

Если применить команду `copper check` к манифесту Kubernetes, который содержит спецификацию образа контейнера `latest`, на экране появится следующее сообщение об ошибке:

```
copper check --rules no_latest.cop --files deployment.yml
Validating part 0
    NoLatest - FAIL
```

Запускать Copper вместе с такими правилами лучше в рамках вашей системы контроля версий (например, чтобы проверять манифесты Kubernetes перед фиксацией кода или проводить автоматические проверки при запросе на внесение изменений).

Еще один аналогичный инструмент, kubeval, сверяет ваши манифесты со спецификацией Kubernetes API (подробнее об этом — в подразделе «kubeval» на с. 298).

kube-bench

kube-bench (github.com/aquasecurity/kube-bench) — это инструмент для аудита кластеров Kubernetes с помощью набора тестов производительности, разработанных Центром интернет-безопасности (Center for Internet Security или CIS). Он, в сущности, проверяет, настроен ли ваш кластер в соответствии с рекомендациями безопасности. Тесты, которые выполняет kube-bench, можно конфигурировать с собственными тестами, написанными в виде документов YAML, или даже дополнять ими (хотя вряд ли вам это пригодится).

Ведение журнала аудита для Kubernetes

Представьте, что вы нашли в своем кластере проблему (например, незнакомую вам pod-оболочку) и хотите понять, откуда она взялась. Как узнать, кто, что и когда делал с вашим кластером? Для этого в Kubernetes существует журнал аудита (kubernetes.io/docs/tasks/debug-application-cluster/audit).

Если включить ведение этого журнала, все запросы к API кластера будут записываться с пометкой о том, когда и кто их сделал (какая служебная учетная запись). Записи будут содержать такие подробности, как запрошенные ресурсы и возвращенный ответ.

Данные о событиях аудита можно направлять в вашу центральную систему ведения журнала с поддержкой фильтрации и оповещений, как любые другие журнальные данные (см. главу 15). Качественные управляемые сервисы, такие как Google Kubernetes Engine, занимаются ведением журнала по умолчанию, а если вы размещаете кластер самостоятельно, вам, возможно, придется включить эту функцию отдельно.

Хаотическое тестирование

В пункте «Доверяй, но проверяй» на с. 68 мы отмечали, что по-настоящему удостовериться в высокой доступности кластера можно только путем удаления одного или нескольких узлов и наблюдения за его дальнейшей работой. То же самое касается высокой доступности pod-оболочек и приложений. Вы, к примеру, можете выбрать произвольную pod-оболочку, принудительно ее остановить и проверить, перезапустит ли ее Kubernetes и не повлияет ли это на частоту ошибок в системе.

Ручное выполнение такой процедуры занимает много времени. К тому же вы можете подсознательно обходить стороной слишком важные для ваших приложений ресурсы. Чтобы проверка была честной, ее следует автоматизировать.

Такой вид автоматического, произвольного вмешательства в промышленные сервисы иногда называют *тестом обезьяны* в честь инструмента Chaos Monkey («хаотическая обезьяна»), разработанного компанией Netflix для тестирования своей инфраструктуры.

> Представьте себе обезьяну, забежавшую в вычислительный центр с серверными фермами, которые обслуживают жизненно важные функции нашей с вами онлайн-деятельности. Обезьяна вырывает первые попавшиеся под руку кабели, ломает устройства…
>
> Задача ИТ-руководителей состоит в том, чтобы информационная система, находящаяся под их ответственностью, могла работать, несмотря на подобных «обезьян», которые могут наведаться в любой момент и уничтожить что угодно.
>
> *Антонио Гарсия Мартинез. Chaos Monkeys*

Инструмент Chaos Monkey, выключающий произвольные облачные серверы, входит в состав пакета Netflix *Simian Army*, который содержит и другие инструменты *хаотического тестирования*. Latency Monkey, например, создает коммуникационные задержки, симулируя проблемы с сетью, Security Monkey ищет известные уязвимости, а Chaos Gorilla выключает целую зону доступности AWS.

Промышленные условия только в промышленной среде

Концепция Chaos Monkey применима и к приложениям Kubernetes. Инструменты хаотического тестирования можно запускать и на проверочном кластере: это позволит не нарушать работу промышленной системы, однако и не даст вам стопроцентной уверенности. Чтобы получить реальные результаты, необходимо тестировать промышленную среду.

> Многие системы слишком большие, сложные и дорогие, чтобы их клонировать. Представьте, что вам для тестирования нужно запустить копию Facebook (которая представляла бы собой несколько глобально распределенных вычислительных центров).
>
> Непредсказуемость пользовательского трафика исключает возможность его имитации; даже если вам удастся идеально воспроизвести то, что происходило вчера, вы все равно не сможете предсказать завтрашние результаты. Промышленные условия только в промышленной среде.
>
> *Чарити Мэйджорс*[1]

Важно отметить, что для максимальной эффективности ваши хаотические эксперименты должны быть автоматизированными и непрерывными. Недостаточно однажды выполнить проверку и рассчитывать на то, что система всегда будет оставаться надежной.

> Весь смысл автоматизации хаотических экспериментов в том, чтобы выполнять их снова и снова, укрепляя уверенность в своей системе. Это нужно не только для выявления новых слабых мест, но и для гарантии того, что вы справились с ранее найденными проблемами.
>
> *Расс Майлз (ChaosIQ)*[2]

Есть несколько инструментов, которые можно использовать для автоматизации хаотического тестирования вашего кластера. Ниже приведем несколько вариантов.

chaoskube

chaoskube (github.com/linki/chaoskube) случайным образом удаляет pod-оболочки вашего кластера. По умолчанию используется пробный режим, когда просто показано, что будет сделано, но на самом деле ничего не удаляется.

Pod-оболочки можно выбирать и исключать по меткам (см. раздел «Метки» на с. 205), аннотациям и пространствам имен; вы также можете указать периоды времени и даты, которых необходимо избегать (чтобы, например, не удалять ничего в рождественскую ночь). По умолчанию удаляются любые pod-оболочки в любых пространствах имен, включая те, что являются частью Kubernetes, и сам процесс chaoskube.

---

1 https://opensource.com/article/17/8/testing-production.

2 medium.com/chaosiq/exploring-multi-level-weaknesses-using-automated-chaos-experiments-aa30f0605ce.

Закончив настраивать фильтр для chaoskube, вы можете выключить пробный режим и позволить этому инструменту делать свою работу.

Программа chaoskube проста в установке и настройке: это идеальное решение для первых шагов в области хаотического тестирования.

kube-monkey

kube-monkey (github.com/asobti/kube-monkey) запускается в заранее установленное время (по умолчанию это восемь утра в рабочие дни) и формирует график развертываний, которые будут объектами тестирования на протяжении всего дня (по умолчанию с 10:00 до 16:00). В отличие от некоторых других инструментов, kube-monkey работает только с явно обозначенными pod-оболочками, содержащими аннотации.

Это означает, что вы можете внедрять тестирование kube-monkey в определенные приложения или сервисы на этапе разработки, устанавливая для них разные частоту и уровень агрессивности. Например, следующая аннотация в pod-оболочке устанавливает среднее время между отказами (mean time between failures, MTBF) продолжительностью два дня:

```
kube-monkey/mtbf: 2
```

Аннотация `kill-mode` позволяет указать, какое количество pod-оболочек в развертывании будет удалено, или их максимальную долю. Следующие аннотации удалят до 50 % pod-оболочек в тестируемом развертывании:

```
kube-monkey/kill-mode: "random-max-percent"
kube-monkey/kill-value: 50
```

PowerfulSeal

PowerfulSeal (github.com/bloomberg/powerfulseal) — это открытый инструмент для хаотического тестирования Kubernetes, который работает в двух режимах: интерактивном и автономном. Интерактивный режим позволяет исследовать кластер и вручную ломать его компоненты, чтобы увидеть, как он отреагирует. Удалять можно узлы, пространства имен, развертывания и отдельные pod-оболочки.

В автономном режиме используется указанный вами набор политик: вы определяете, с какими ресурсами работать, каких избегать, когда запускать (например, можно сделать так, чтобы он действовал только в рабочее время с понедельника по пятницу) и насколько агрессивно нужно себя вести (например, удалить заданный процент от всех подходящих развертываний). Файлы с политиками Kubernetes

очень гибкие и позволяют подготовить практически любой сценарий хаотического тестирования, который только можно вообразить.

Рекомендуемый подход

Если ваши приложения должны быть высокодоступными, инструменты хаотического тестирования, такие как chaoskube, следует запускать регулярно. Это даст уверенность в том, что неожиданный отказ узла или pod-оболочки не вызовет проблем. Обязательно обсудите это с людьми, ответственными за работу кластера и приложений, которые тестируются.

Резюме

Подбор размера и конфигурации ваших первых кластеров Kubernetes может оказаться очень сложной задачей. Доступно много вариантов, и пока вы не получите опыт работы в промышленных условиях, вы не узнаете, какой из них вам действительно подходит.

Мы не можем принимать решения за вас, но надеемся, что в этой главе вы получили достаточно информации, которая поможет сделать выбор.

- Прежде чем создавать промышленный кластер Kubernetes, подумайте о том, сколько узлов вам понадобится и насколько большими они должны быть.
- Вам нужно как минимум три ведущих узла (в управляемом сервисе не нужно ни одного) и по меньшей мере два рабочих (в идеале три). На первый взгляд может показаться, что кластеры Kubernetes станут более дорогими (особенно если выполнять только несколько небольших рабочих заданий), но не забывайте о преимуществах встроенных устойчивости и масштабирования.
- Кластеры Kubernetes могут масштабироваться на многие тысячи узлов и сотни тысяч контейнеров.
- Если такого масштабирования вам недостаточно, используйте сразу несколько кластеров (иногда это требуется по причинам безопасности или для соблюдения правовых норм). Кластеры можно объединять в федерацию, если есть необходимость распределять между ними свои рабочие задания.
- Типичный сервер для узла Kubernetes имеет один процессор и 4 ГиБ оперативной памяти. Хотя лучше сочетать узлы разных размеров.
- Платформа Kubernetes не только подходит для облака, но и работает на физических серверах. Если у вас завалялось лишнее оборудование, почему бы им не воспользоваться?

- Кластер можно расширять и сокращать вручную без особых усилий, тем более что делать это, скорее всего, придется не очень часто. Автомасштабирование — хорошая, но не такая уж и важная возможность.
- Для поставщиков и продуктов на основе Kubernetes существует четко определенный стандарт: знак качества *Certified Kubernetes* от CNCF. Если вы его не видите, поинтересуйтесь, почему его нет.
- Хаотическое тестирование — процесс удаления произвольных pod-оболочек с последующим наблюдением за работой приложения. Это полезная методика, но у облачных платформ есть свои собственные механизмы подобного рода, которые делают все за вас.

7 Продвинутые инструменты для работы с Kubernetes

> Мой механик сказал: «Я не смог починить ваши тормоза, поэтому сделал громче ваш гудок».
>
> *Стивен Райт*

Нас постоянно спрашивают: «А что насчет всех этих инструментов для Kubernetes? Нужны ли они нам? Если да, то какие именно? И для чего они все предназначены?»

В этой главе мы исследуем лишь небольшую часть инструментов и утилит, которые помогают в работе с Kubernetes: покажем некоторые продвинутые функции `kubectl` и представим несколько полезных утилит, таких как `jq`, `kubectx`, `kubens`, `kube-ps1`, `kubeshell`, Click, `kubed-sh`, Stern и BusyBox.

Осваиваем kubectl

Знакомство с `kubectl` мы начали еще в главе 2. Это главный инструмент для взаимодействия с Kubernetes, поэтому, скорее всего, вам уже известны его основные положения. А теперь рассмотрим некоторые из наиболее продвинутых возможностей `kubectl`, а также дадим советы и рекомендации, которые могут быть для вас новыми.

Псевдонимы командной оболочки

Чтобы сделать свою жизнь проще, большинство пользователей Kubernetes первым делом создают псевдоним командной оболочки для `kubectl`. Например, в наших файлах `.bash_profile` указан следующий псевдоним:

```
alias k=kubectl
```

Теперь вместо того, чтобы полностью вводить `kubectl` в каждой команде, мы можем использовать `k`:

```
k get pods
```

Если вы часто используете какие-то команды `kubectl`, создайте псевдонимы и для них. Ниже показаны возможные примеры:

```
alias kg=kubectl get
alias kgdep=kubectl get deployment
alias ksys=kubectl --namespace=kube-system
alias kd=kubectl describe
```

Один из инженеров Google Ахмед Альп Балкан разработал логическую систему подобных псевдонимов (ahmet.im/blog/kubectl-aliases/index.html) и написал скрипт, который может сгенерировать их для вас (на сегодня доступно около 800 вариантов).

Но вам не обязательно их использовать: советуем начать с `k` и затем добавлять запоминающиеся сокращения для тех команд, которые вы вводите чаще всего.

Использование коротких флагов

Как и большинство инструментов командной строки, `kubectl` поддерживает краткие формы многих своих флагов и переключателей. Это существенно сокращает время на их ввод.

Например, флаг `--namespace` можно сократить до `-n` (см. раздел «Использование пространств имен» на с. 117):

```
kubectl get pods -n kube-system
```

`kubectl` очень часто используют для работы с ресурсами, которые соответствуют какому-то набору меток. Для этого предусмотрен флаг `--selector` (см. раздел «Метки» на с. 205). К счастью, его можно сократить до `-l` (от англ. labels):

```
kubectl get pods -l environment=staging
```

Сокращение названий типов ресурсов

С помощью `kubectl` часто выводят список ресурсов различных типов, таких как pod-оболочки, развертывания, сервисы и пространства имен. Обычно используется команда `kubectl get`, за которой идет, к примеру, `deployments`.

Чтобы ускорить данный процесс, `kubectl` поддерживает краткие названия этих типов ресурсов:

```
kubectl get po
kubectl get deploy
kubectl get svc
kubectl get ns
```

Среди других полезных сокращений можно выделить `cm` для `configmaps`, `sa` для `serviceaccounts`, `ds` для `daemonsets` и `pv` для `persistentvolumes`.

Автодополнение команд kubectl

Если вы используете командные оболочки `bash` или `zsh`, то можете сделать так, чтобы они автоматически дополняли команды `kubectl`. Как это делается в вашей оболочке, узнаете, выполнив следующую команду:

```
kubectl completion -h
```

В итоге, если следовать инструкциям, по нажатию клавиши Tab ваши команды `kubectl` будут дополняться. Попробуйте сами:

```
kubectl cl<TAB>
```

Эта команда должна превратиться в `kubectl cluster-info`.

Если введете `kubectl` и дважды нажмете Tab, увидите все доступные команды:

```
kubectl <TAB><TAB>
alpha           attach       cluster-info   cordon     describe   ...
```

Тот же подход можно использовать для вывода всех флагов текущей команды:

```
kubectl get pods --<TAB><TAB>
--all-namespaces   --cluster=   --label-columns=   ...
```

`kubectl` также автоматически дополняет названия pod-оболочек, развертываний, пространств имен и т. д.:

```
kubectl -n kube-system describe pod <TAB><TAB>
event-exporter-v0.1.9-85bb4fd64d-2zjng
kube-dns-autoscaler-79b4b844b9-2wglc
fluentd-gcp-scaler-7c5db745fc-h7ntr
...
```

Справка

Лучшие инструменты командной строки поставляются вместе с подробной документацией, и `kubectl` не является исключением. Чтобы получить полный обзор доступных команд, введите `kubectl -h`:

```
kubectl -h:
```

Вы можете пойти дальше и получить подробное описание каждой команды со всеми доступными параметрами и примерами. Для этого надо ввести `kubectl COMMAND -h`:

```
kubectl get -h
```

Справка по ресурсам Kubernetes

Помимо документации по своим собственным возможностям, `kubectl` предоставляет справку и для объектов Kubernetes, таких как развертывания или pod-оболочки. Команда `kubectl explain` выводит документацию для заданного типа ресурсов:

```
kubectl explain pods
```

Вы можете получить дополнительную информацию об определенном поле ресурса с помощью команды `kubectl explain RESOURCE.FIELD`. На самом деле `explain` позволяет копнуть настолько глубоко, насколько вы захотите:

```
kubectl explain deploy.spec.template.spec.containers.livenessProbe.exec
```

Как вариант, можете попробовать команду `kubectl explain --recursive`, которая выводит поля внутри полей внутри других полей... смотрите, а то закружится голова!

Отображение более подробного вывода

Вы уже знаете, что `kubectl get` выводит список ресурсов различных типов, таких как pod-оболочки:

```
kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
demo-54f4458547-pqdxn    1/1     Running   6          5d
```

С помощью флага `-o wide` можно получить дополнительную информацию, например, о том, на каком узле запущена каждая pod-оболочка:

```
kubectl get pods -o wide
NAME                     ... IP           NODE
demo-54f4458547-pqdxn    ... 10.76.1.88   gke-k8s-cluster-1-n1-standard...
```

Чтобы сэкономить место, мы убрали часть вывода, которую можно видеть без флага `-o wide`.

В зависимости от типа ресурса флаг `-o wide` показывает разную информацию. Например, в случае с узлами это выглядит так:

```
kubectl get nodes -o wide
NAME              ... EXTERNAL-IP      OS-IMAGE        KERNEL-VERSION
gke-k8s-...8l6n   ... 35.233.136.194   Container...    4.14.22+
gke-k8s-...dwtv   ... 35.227.162.224   Container...    4.14.22+
gke-k8s-...67ch   ... 35.233.212.49    Container...    4.14.22+
```

Работа с jq и данными в формате JSON

По умолчанию команда `kubectl get` возвращает данные в виде обычного текста, но вы можете вывести их в формате JSON:

```
kubectl get pods -n kube-system -o json
{
     "apiVersion": "v1",
     "items": [
     {
       "apiVersion": "v1",
       "kind": "Pod",
       "metadata": {
           "creationTimestamp": "2018-05-21T18:24:54Z",
           ...
```

Неудивительно, что результат получился таким большим (около 5000 строчек на нашем кластере). К счастью, мы имеем дело с широко используемым форматом JSON, и поэтому его можно отфильтровать с помощью других инструментов, таких как бесценная утилита `jq`.

Если у вас нет утилиты `jq` (stedolan.github.io/jq/manual), установите ее принятым в вашей системе способом (`brew install jq` для macOS, `apt install jq` для Debian/Ubuntu и т. д.).

Закончив с установкой, вы можете использовать `jq` для выполнения запросов и фильтрации вывода `kubectl`:

```
kubectl get pods -n kube-system -o json | jq '.items[].metadata.name'
"event-exporter-v0.1.9-85bb4fd64d-2zjng"
"fluentd-gcp-scaler-7c5db745fc-h7ntr"
"fluentd-gcp-v3.0.0-5m627"
"fluentd-gcp-v3.0.0-h5fjg"
...
```

`jq` является очень мощным инструментом для запрашивания и трансформации данных в формате JSON.

Например, вы можете перечислить самые занятые — по количеству pod-оболочек — узлы:

```
kubectl get pods -o json --all-namespaces | jq '.items |
  group_by(.spec.nodeName) | map({"nodeName": .[0].spec.nodeName,
  "count": length}) | sort_by(.count) | reverse'
```

Для `jq` создана онлайн-площадка (jqplay.org), где вы можете вставлять JSON-данные и применять к ним различные запросы, чтобы получить именно тот результат, который вам нужен.

На случай, если у вас нет доступа к `jq`, `kubectl` также поддерживает запросы JSONPath (kubernetes.io/docs/reference/kubectl/jsonpath). JSONPath — это язык запросов для JSON, который уступает по своим возможностям `jq`, но удобен для быстрого написания однострочных селекторов:

```
kubectl get pods -o=jsonpath={.items[0].metadata.name}
demo-66ddf956b9-pnknx
```

Наблюдение за объектами

Когда вы ждете запуска большого количества pod-оболочек, набирать `kubectl get pods...` каждые несколько секунд, чтобы проверить результат, может быстро надоесть.

У `kubectl` есть флаг `--watch` (сокращенно `-w`), который избавит вас от этого. Например:

```
kubectl get pods --watch
NAME                   READY   STATUS              RESTARTS   AGE
demo-95444875c-z9xv4   0/1     ContainerCreating   0          1s
... [time passes] ...
demo-95444875c-z9xv4   0/1     Completed           0          2s
demo-95444875c-z9xv4   1/1     Running             0          2s
```

Каждый раз, когда меняется состояние одной из подходящих pod-оболочек, вывод в терминале будет обновляться (в подразделе «Наблюдение за ресурсами Kubernetes с помощью kubespy» на с. 168 показан изящный способ наблюдения за любыми видами ресурсов).

Описание объектов

Чтобы получить по-настоящему подробную информацию об объектах Kubernetes, можно воспользоваться командой `kubectl describe`:

```
kubectl describe pods demo-d94cffc44-gvgzm
```

Раздел `Events` особенно полезен для отладки контейнеров, которые не работают должным образом, поскольку в нем записан каждый этап жизненного цикла контейнера вместе с любыми возникшими ошибками.

Работа с ресурсами

До сих пор мы использовали `kubectl` для выполнения запросов и вывода информации, а также для применения декларативных YAML-манифестов с помощью `kubectl apply`. Но у `kubectl` также есть полный набор *императивных* команд: операций для непосредственного создания или редактирования ресурсов.

Императивные команды kubectl

Один из примеров был показан в подразделе «Запуск демонстрационного приложения» на с. 61, где мы использовали команду `kubectl run`, которая автоматически создает развертывание для выполнения в заданном контейнере.

Большинство ресурсов можно создавать явным образом с помощью команды `kubectl create`:

```
kubectl create namespace my-new-namespace
namespace "my-new-namespace" created
```

Аналогично `kubectl delete` и удалит ресурс:

```
kubectl delete namespace my-new-namespace
namespace "my-new-namespace" deleted
```

Команда `kubectl edit` дает вам возможность просматривать и модифицировать любые ресурсы:

```
kubectl edit deployments my-deployment
```

Этим вы откроете свой стандартный текстовый редактор с файлом манифеста в формате YAML, который представляет заданный ресурс.

Это хороший способ получить детальную информацию о конфигурации любого ресурса, но также вы можете в редакторе внести и необходимые изменения. После сохранения файла и выхода из редактора `kubectl` обновит ресурс так, словно вы применили к манифесту ресурса команду `kubectl apply`.

Если вы допустили какие-либо ошибки (например, некорректный формат YAML), `kubectl` об этом сообщит и откроет файл, чтобы вы могли исправить проблему.

Когда не следует использовать императивные команды

В этой книге мы постоянно подчеркиваем важность использования *декларативной* инфраструктуры в качестве кода. Поэтому неудивительно, что мы не советуем прибегать к императивным командам `kubectl`.

Они могут быть крайне полезными для быстрого тестирования или проверки новых идей, но их основная проблема в том, что у вас нет единого *источника истины*. Вы не можете определить, кто и когда запускал те или иные команды в кластере и к каким результатам это привело. Как только вы выполните императивную команду, состояние вашего кластера перестанет быть синхронизированным с файлами манифестов, хранящимися в системе контроля версий.

В следующий раз, когда кто-нибудь применит YAML-манифесты, все изменения, которые вы внесли императивным образом, будут перезаписаны и утеряны. Это может привести к неожиданным последствиям и вызвать потенциально нежелательные эффекты для критически важных сервисов.

> Во время дежурства Элис неожиданно происходит большой скачок нагрузки на сервис, которым она занимается. Элис использует команду kubectl scale, чтобы увеличить количество реплик с пяти до десяти. Несколькими днями позже Боб редактирует YAML-манифест в системе контроля версий, чтобы перейти на новый образ контейнеров, и при этом не замечает, что текущее число реплик в файле равно пяти, хотя в промышленной среде их десять. Боб выкатывает изменения, чем уменьшает число реплик вдвое и вызывает немедленную перегрузку или перебой в работе.
>
> *Келси Хайтауэр и др. Kubernetes Up & Running*

Элис забыла обновить файлы в системе контроля версий после внесения императивных изменений. Такую ошибку легко допустить, особенно под давлением (см. подраздел «Дежурство не должно быть пыткой» на с. 368): в реальной жизни не все проходит гладко.

Так же и Боб перед повторным применением файлов манифеста должен был проверить изменения с помощью команды `kubectl diff` (см. подраздел «Сравнение ресурсов» на с. 166). Ведь если вы не ожидаете, что изменения были, несложно и упустить это из виду. К тому же Боб, наверное, не читал нашу книгу.

Лучший способ избежать подобного рода проблем — всегда вносить изменения путем редактирования и применения файлов ресурсов в системе контроля версий.

Рекомендуемый подход

Не используйте императивные команды kubectl, такие как create или edit, в промышленных кластерах. Вместо этого всегда управляйте ресурсами с помощью YAML-манифестов в системе контроля версий и применяйте их командой kubectl apply (или используя чарты Helm).

Генерация манифестов ресурсов

Несмотря на то что мы не советуем использовать `kubectl` в интерактивном режиме для внесения изменений в ваш кластер, императивные команды могут сэкономить немало времени при создании с нуля YAML-файлов для Kubernetes.

Вместо того чтобы набирать огромные шаблонные фрагменты в пустом файле, вы можете сгенерировать YAML-манифест с помощью `kubectl`:

```
kubectl run demo --image=cloudnatived/demo:hello --dry-run -o yaml
apiVersion: extensions/v1beta1
kind: Deployment
...
```

Флаг `--dry-run` говорит `kubectl` о том, что вместо создания самого ресурса `kubectl` следует вывести его манифест. Флаг `-o yaml` позволяет отобразить манифест ресурса в формате YAML. Вы можете сохранить этот вывод в файл, отредактировать его (если требуется) и применить для создания ресурса в кластере:

```
kubectl run demo --image=cloudnatived/demo:hello --dry-run -o yaml
    >deployment.yaml
```

Итак, внесите какие-нибудь правки с помощью предпочитаемого вами редактора, сохраните и примените результат:

```
kubectl apply -f deployment.yaml
deployment.apps "demo" created
```

Экспорт ресурсов

`kubectl` может помочь с созданием манифестов не только для новых, но и для уже существующих ресурсов. Представьте, к примеру, что вы создали развертывание с помощью интерактивной команды (`kubectl run`), затем отредактировали его, установив подходящие настройки, и теперь вам нужно написать для него декларативный манифест в формате YAML, который можно будет добавить в систему контроля версий.

Чтобы это сделать, укажите для команды `kubectl get` флаг `--export`:

```
kubectl run newdemo --image=cloudnatived/demo:hello --port=8888
--labels app=newdemo
deployment.apps "newdemo" created
kubectl get deployments newdemo -o yaml --export >deployment.yaml
```

Этот вывод имеет формат, позволяющий сохранить данный манифест вместе с другими манифестами, а затем обновить его и применить с помощью команды `kubectl apply -f`.

Если до сих пор вы использовали для управления своим кластером интерактивные команды `kubectl`, но хотите перейти на рекомендуемый нами декларативный стиль, есть отличный способ: экспортируйте все ресурсы своего кластера в файлы манифестов, используя команду `kubectl` с флагом `--export` (как мы показывали в нашем примере), и все будет готово.

Сравнение ресурсов

Прежде чем применять манифесты Kubernetes с помощью команды `kubectl apply`, было бы крайне полезно увидеть, что же на самом деле изменится в кластере. Для этого предусмотрена команда `kubectl diff`:

```
kubectl diff -f deployment.yaml
-  replicas: 10
+  replicas: 5
```

Благодаря этому выводу можно убедиться, что вносимые вами изменения приведут именно к тем результатам, которых вы ожидали. Вы также будете предупреждены, если состояние активного ресурса рассинхронизировано с YAML-манифестом — возможно, с того момента, когда вы применяли манифест в последний раз, кто-то отредактировал его императивным способом.

Рекомендуемый подход

Чтобы проверить потенциальные изменения перед тем, как применить их к своему промышленному кластеру, используйте kubectl diff.

Работа с контейнерами

Большая часть работы кластера Kubernetes происходит внутри контейнеров, поэтому, если что-то идет не так, разобраться бывает непросто. Ниже представлено несколько способов работы с запущенными контейнерами с помощью `kubectl`.

Просмотр журнальных записей контейнера

Если вы пытаетесь привести контейнер к рабочему состоянию, а он ведет себя не так, как вам бы хотелось, одним из самых полезных источников информации являются его журнальные записи. С точки зрения Kubernetes *журналом* считается все, что контейнер записывает в *потоки вывода сообщений об ошибках* — это то, что выводит на экран программа, запущенная в терминале.

При работе с промышленными, в особенности распределенными, приложениями у вас должна быть возможность агрегировать журнальные записи нескольких сервисов, сохранять их в постоянной базе данных, выполнять к ним запросы и визуализировать их. Это обширная тема, и мы обсудим ее намного детальнее в главе 15.

Но исследование журнальных сообщений отдельных контейнеров по-прежнему очень полезно для отладки. Сделать это можно непосредственно с помощью команды `kubectl logs`, указав ей имя pod-оболочки:

```
kubectl logs -n kube-system --tail=20 kube-dns-autoscaler-69c5cbdcdd-94h7f
autoscaler.go:49] Scaling Namespace: kube-system, Target: deployment/kube-dns
autoscaler_server.go:133] ConfigMap not found: configmaps "kube-dns-autoscaler"
k8sclient.go:117] Created ConfigMap kube-dns-autoscaler in namespace kube-system
plugin.go:50] Set control mode to linear
linear_controller.go:59] ConfigMap version change (old:  new: 526) - rebuilding
```

Большинство долгоработающих контейнеров генерируют *огромный* журнал, поэтому вам лучше ограничиться лишь самыми последними строчками. Для этого в данном примере используется флаг `--tail` (журнальный вывод контейнера содержит временные метки, но мы их опустили, чтобы уместить вывод на странице).

Чтобы следить за контейнером и направлять его журнальный вывод в терминал, используйте флаг `--follow` (сокращенно `-f`):

```
kubectl logs --namespace kube-system --tail=10 --follow etcd-docker-for-desktop
etcdserver: starting server... [version: 3.1.12, cluster version: 3.1]
embed: ClientTLS: cert = /var/lib/localkube/certs/etcd/server.crt, key = ...
...
```

Если позволить команде `kubectl logs` работать дальше, вы сможете увидеть вывод из контейнера `etcd-docker-for-desktop`.

Особенно полезно будет просмотреть журнал сервера API Kubernetes: например, если возникнут ошибки с правами доступа RBAC (см. подраздел «Введение в управление доступом на основе ролей» на с. 258), вы сможете их там найти. При наличии доступа к ведущим узлам вы можете найти pod-оболочку `kube-apiserver` в пространстве имен `kube-system` и воспользоваться командой `kubectl logs`, чтобы увидеть ее вывод.

Если вы используете управляемый сервис вроде GKE, где не видны ведущие узлы, проверьте документацию вашего поставщика, чтобы понять, как найти журнальные записи управляющего уровня (в GKE, например, они показываются в Stackdriver Logs Viewer).

Если в pod-оболочке есть несколько контейнеров, с помощью флага --container (сокращенно -c) можно указать, журнальные записи которого из них вы хотите увидеть:

```
kubectl logs -n kube-system metrics-server
    -c metrics-server-nanny
...
```

Для более гибкого слежения за журналами лучше использовать отдельный инструмент, такой как Stern (см. подраздел «Stern» на с. 178).

Подключение к контейнеру

Если просмотра журнальных записей контейнера недостаточно, возможно, вы захотите подключить к нему свой локальный терминал. Это позволит наблюдать вывод контейнера напрямую. Для этого предусмотрена команда `kubectl attach`:

```
kubectl attach demo-54f4458547-fcx2n
Defaulting container name to demo.
Use kubectl describe pod/demo-54f4458547-fcx2n to see all of the containers
in this pod.
If you don't see a command prompt, try pressing enter.
```

Наблюдение за ресурсами Kubernetes с помощью kubespy

Когда вы вносите изменения в манифесты Kubernetes, часто бывает тревожный период ожидания того, что произойдет дальше.

При развертывании приложения много интересного остается скрытым от глаз: Kubernetes создает ваши ресурсы, запускает pod-оболочки и т. д.

Поскольку все происходит *автомагически*, как любят говорить инженеры, разобраться в этом бывает непросто. Команды `kubectl get` и `kubectl describe` выводят

статическое состояние отдельных ресурсов, но хотелось бы видеть процесс в режиме реального времени.

Представляем вашему вниманию kubespy (github.com/pulumi/kubespy), удобный инструмент от проекта Pulumi[1]. Он может наблюдать за отдельными ресурсами кластера и показывать вам, что с ними со временем происходит.

Например, если направить kubespy на сервис, вы узнаете, когда тот был создан, когда ему был выделен IP-адрес, когда были подключены его конечные точки и т. д.

Перенаправление порта контейнера

Мы уже использовали команду `kubectl port-forward` в подразделе «Запуск демонстрационного приложения» на с. 61 для привязки сервиса Kubernetes к порту на вашем локальном компьютере. Но с ее помощью также можно перенаправить порт контейнера, если вы хотите подключиться напрямую к определенной pod-оболочке. Просто укажите имя оболочки и локальный и удаленный порты:

```
kubectl port-forward demo-54f4458547-vm88z 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

Теперь порт 9999 на вашем локальном компьютере будет направлен к порту 8888 контейнера, и вы сможете подключиться к нему, например, из браузера.

Выполнение команд внутри контейнеров

Изолированность контейнеров отлично подходит для выполнения надежных и безопасных рабочих заданий. Но если что-то не работает, а вы не понимаете почему, она может создавать небольшие неудобства.

Когда вы запускаете программу на своем локальном компьютере и она начинает вести себя странно, в вашем распоряжении находится вся мощь командной строки: вы можете просмотреть текущие процессы с помощью `ps`, вывести содержимое папок и файлов, используя `ls` и `cat`, и даже отредактировать их в `vi`.

В случае с неисправным контейнером очень часто было бы полезно иметь внутри него запущенную командную оболочку для проведения такого рода интерактивной отладки.

---

[1] Pulumi — это облачно-ориентированный фреймворк в стиле «инфраструктура как код», www.pulumi.com.

С помощью `kubectl exec` в любом контейнере можно запустить любую команду, включая командную оболочку:

```
kubectl run alpine --image alpine --command -- sleep 999
deployment.apps "alpine" created

kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
alpine-7fd44fc4bf-7gl4n   1/1     Running   0          4s

kubectl exec -it alpine-7fd44fc4bf-7gl4n /bin/sh
/ # ps
PID   USER     TIME   COMMAND
    1 root       0:00 sleep 999
    7 root       0:00 /bin/sh
   11 root       0:00 ps
```

Если pod-оболочка содержит больше одного контейнера, `kubectl exec` по умолчанию выполняет команду только в первом из них. Но вы также можете указать нужный вам контейнер с помощью флага `-c`:

```
kubectl exec -it -c container2 POD_NAME /bin/sh
```

Если у контейнера нет командной оболочки, см. подраздел «Добавление BusyBox в ваш контейнер» на с. 172.

Запуск контейнеров с целью отладки

Помимо запуска команд в существующих контейнерах, иногда бывает полезно запускать утилиты вроде `wget` или `nslookup` в самом кластере, чтобы увидеть, какие результаты получит ваше приложение. Вы уже знаете, как запускать контейнеры в кластере с использованием `kubectl run`; ниже приводится несколько примеров с запуском одноразовых контейнерных команд с целью отладки.

Для начала попробуем запустить экземпляр нашего демонстрационного приложения:

```
kubectl run demo --image cloudnatived/demo:hello --expose --port 8888
service "demo" created
deployment.apps "demo" created
```

Сервис `demo` должен получить IP-адрес и доменное имя (`demo`), доступные изнутри кластера. Убедимся в этом, запустив команду `nslookup` в контейнере:

```
kubectl run nslookup --image=busybox:1.28 --rm --it --restart=Never \
--command -- nslookup demo
Server:    10.79.240.10
```

```
Address 1: 10.79.240.10 kube-dns.kube-system.svc.cluster.local

Name:      demo
Address 1: 10.79.242.119 demo.default.svc.cluster.local
```

Хорошие новости: доменное имя работает, поэтому у нас должна быть возможность отправлять ему HTTP-запросы с помощью `wget`. Посмотрим, что получим в ответ:

```
kubectl run wget --image=busybox:1.28 --rm --it --restart=Never \
--command -- wget -qO- http://demo:8888
Hello, 世界
```

Как видите, команды типа `kubectl run` имеют один и тот же набор флагов:

```
kubectl run NAME --image=IMAGE --rm --it --restart=Never --command -- ...
```

Зачем они нужны?

- `--rm`. Этот флаг говорит Kubernetes о необходимости удалить образ контейнера после завершения его работы, чтобы тот не занимал место в локальном хранилище узла.
- `--it`. Запускает контейнер интерактивно (`i` — interactively) в локальном терминале (`t` — terminal), чтобы вы могли просматривать его вывод и при необходимости отправлять ему информацию о нажатых клавишах.
- `--restart=Never`. Говорит Kubernetes не перезапускать контейнер каждый раз, когда тот завершает работу (поведение по умолчанию). Мы можем отключить стандартную политику перезапуска, так как нам нужно запустить контейнер лишь один раз.
- `--command--`. Вместо точки входа контейнера по умолчанию указывает команду, которую нужно выполнить. Все, что идет за `--`, будет передано контейнеру в виде командной строки, включая аргументы.

Использование команд BusyBox

Вы можете запускать любые доступные вам контейнеры, однако образ `busybox` является особенно полезным ввиду наличия в нем большого количества самых востребованных в Unix команд, таких как `cat`, `echo`, `find`, `grep` и `kill`. Их полный список можно просмотреть на сайте BusyBox (busybox.net/downloads/BusyBox.html).

BusyBox также включает в себя маловесящую командную оболочку в стиле bash. Она называется ash и имеет совместимость со стандартными командными скриптами `/bin/sh`. Итак, чтобы получить интерактивную командную оболочку в своем кластере, выполните:

```
kubectl run busybox --image=busybox:1.28 --rm --it --restart=Never /bin/sh
```

Поскольку шаблон запуска команд из образа BusyBox никогда не меняется, вы можете даже сделать для него псевдоним (см. подраздел «Псевдонимы командной оболочки» на с. 157):

```
alias bb=kubectl run busybox --image=busybox:1.28 --rm --it --restart=Never
   --command--
bb nslookup demo
...
bb wget -qO- http://demo:8888
...
bb sh
If you don't see a command prompt, try pressing enter.
/ #
```

Добавление BusyBox в ваш контейнер

Если у вашего контейнера уже есть командная оболочка (например, он построен из базового образа Linux, такого как `alpine`), вы можете получить доступ к ней, выполнив:

```
kubectl exec -it POD /bin/sh
```

Но что, если в контейнере нет `/bin/sh`? Как, например, описывается в подразделе «Что собой представляет Dockerfile» на с. 56 — использование минимального образа, созданного с нуля.

Самый простой способ сделать отладку контейнера легкой и сохранить при этом его очень маленький размер — скопировать в него исполняемый файл `busybox` во время сборки. Он занимает всего 1 МиБ, что кажется небольшой ценой за наличие полноценной командной оболочки и набора Unix-утилит.

Во время нашего обсуждения многоэтапной сборки вы научились копировать файлы из ранее собранного контейнера в новый, используя команду `COPY --from` для Dockerfile. Другая, менее известная возможность этой команды позволяет скопировать файл из любого публичного образа, а не только из того, который вы собрали локально.

В следующем примере Dockerfile показано, как это сделать для образа `demo`:

```
FROM golang:1.11-alpine AS build

WORKDIR /src/
COPY main.go go.* /src/
RUN CGO_ENABLED=0 go build -o /bin/demo

FROM scratch
COPY --from=build /bin/demo /bin/demo
COPY --from=busybox:1.28 /bin/busybox /bin/busybox
ENTRYPOINT ["/bin/demo"]
```

Флаг `--from=busybox:1.28` ссылается на публичный библиотечный образ BusyBox[1]. Но вы можете копировать файлы из любых образов на ваш выбор (например, из `alpine`).

Ваш контейнер остался очень маленьким, но теперь вы можете получить в нем командную оболочку, запустив:

```
kubectl exec -it POD_NAME /bin/busybox sh
```

Вместо непосредственного выполнения `/bin/sh` вы обращаетесь к `/bin/busybox` и указываете имя нужной вам команды: в данном случае это `sh`.

Установка программ в контейнер

Если вам нужны какие-то программы, не входящие в BusyBox или недоступные в публичных образах контейнеров, можете запустить образ Linux — например, `alpine` или `ubuntu` — и просто установить в нем то, что хотите:

```
kubectl run alpine --image alpine --rm --it --restart=Never /bin/sh
If you don't see a command prompt, try pressing enter.
/ # apk --update add emacs
```

Отладка в режиме реального времени с помощью kubesquash

В этой главе мы упоминали *отладку* контейнеров в довольно широком смысле, имея в виду *выяснение того, что с ними не так*. Ну а если, например, вы хотите подключить к одному из активных процессов в контейнере настоящий отладчик, такой как как `gdb` (отладчик от проекта GNU) или `dlv` (отладчик для Go)?

Отладчики, подобные `dlv`, являются очень мощными инструментами. Они могут подключиться к процессу, показать, какие строчки исходного кода сейчас выполняются, вывести и изменить значения локальных переменных, создать точки останова и пройтись по коду строка за строкой. Если происходит что-то загадочное, с чем вы не можете разобраться, вполне возможно, что рано или поздно вам придется обратиться к отладчику.

При запуске программы на локальном компьютере вы имеете прямой доступ к ее процессам, поэтому проблемы не возникает. Но если программа в контейнере, то все, как часто бывает, становится немного сложнее.

---

[1] У версий BusyBox новее 1.28 имеется проблема с выполнением DNS-запросов в Kubernetes, github.com/kubernetes/kubernetes/issues/66924.

Инструмент kubesquash создан для того, чтобы помочь вам подключить отладчик к контейнеру. Для установки отладчика следуйте инструкциям (github.com/solo-io/kubesquash) в GitHub.

После остается лишь предоставить kubesquash имя активного контейнера:

```
/usr/local/bin/kubesquash-osx demo-6d7dff895c-x8pfd
? Going to attach dlv to pod demo-6d7dff895c-x8pfd. continue? Yes
If you don't see a command prompt, try pressing enter.
(dlv)
```

kubesquash автоматически создаст в пространстве имен squash pod-оболочку с исполняемым файлом отладчика и позаботится о его подключении к процессу, запущенному в заданной вами pod-оболочке.

По техническим причинам (github.com/solo-io/kubesquash/blob/master/cmd/kubesquash/main.go#L13) для работы kubesquash в целевом контейнере должна быть доступна команда ls. Если вы построили контейнер с нуля, можете добавить в него исполняемый файл BusyBox, как мы это делали в подразделе «Добавление BusyBox в ваш контейнер» на с. 172:

```
COPY --from=busybox:1.28 /bin/busybox /bin/ls
```

Вместо /bin/busybox копируем исполняемый файл в /bin/ls: это обеспечивает идеальную работу kubesquash.

Не станем здесь углубляться в подробности использования dlv, но скажем, что это незаменимый инструмент, если вы пишете приложения на Go для Kubernetes: kubesquash позволяет легко применять dlv для контейнеров.

Больше о dlv можно почитать в официальной документации (github.com/go-delve/delve/tree/master/Documentation).

Контексты и пространства имен

До сих пор в этой книге мы работали с единственным кластером Kubernetes, и все команды kubectl, которые выполнялись, естественно, применялись именно к нему.

Но что, если кластеров несколько? Например, помимо кластера Kubernetes для локального тестирования, который находится на вашем компьютере, у вас может быть промышленный облачный кластер и, вполне вероятно, еще один удаленный

для финального тестирования и разработки. Откуда kubectl будет знать, какой из них имеется в виду?

Для решения этой проблемы kubectl предлагает концепцию *контекстов*. Контекст — это сочетание кластера, пользователя и пространства имен (см. раздел «Использование пространств имен» на с. 117).

Когда вы запускаете команды kubectl, они всегда выполняются в *текущем контексте*. Рассмотрим пример:

```
kubectl config get-contexts
CURRENT   NAME                 CLUSTER          AUTHINFO       NAMESPACE
          gke                  gke_test_us-w    gke_test_us    myapp
*         docker-for-desktop   docker-for-d     docker-for-d
```

Это контексты, о которых kubectl в настоящее время знает. Каждый контекст имеет имя и ссылается на определенный кластер, имя пользователя, который в нем аутентифицирован, и пространство имен внутри этого кластера. Контекст docker-for-desktop, как и можно было ожидать, ссылается на наш локальный кластер Kubernetes.

Текущий контекст помечен звездочкой * в первом столбце (в данном примере это dockerfor-desktop). Если мы сейчас запустим команду kubectl, она будет выполнена в пространстве имен по умолчанию кластера Docker Desktop (пустой столбец NAMESPACE говорит о том, что контекст ссылается на пространство имен по умолчанию):

```
kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at https://192.168.99.100:8443/api/v1/...

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

С помощью команды kubectl config use-context можно переключиться на другой контекст:

```
kubectl config use-context gke
Switched to context "gke".
```

Контексты — это своего рода закладки, позволяющие легко переходить в определенное пространство имен определенного кластера. Чтобы создать новый контекст, выполните kubectl config set-context:

```
kubectl config set-context myapp --cluster=gke --namespace=myapp
Context "myapp" created.
```

Теперь, если переключиться в контекст `myapp`, можно работать с пространством имен `myapp` в кластере Docker Desktop.

Если вы забыли, в каком контексте находитесь, команда `kubectl config current-context` напомнит:

```
kubectl config current-context
myapp
```

kubectx и kubens

Если вы, как и мы, зарабатываете на жизнь, набирая текст, вероятно, вам не очень хочется лишний раз нажимать клавиши. Для более быстрого переключения между контекстами `kubectl` можно использовать инструменты `kubectx` и `kubens`. Чтобы их установить, следуйте инструкциям (github.com/ahmetb/kubectx) в GitHub.

Теперь вы можете переключать контексты с помощью команды `kubectx`:

```
kubectx docker-for-desktop
Switched to context "docker-for-desktop".
```

Одной из интересных возможностей этой команды является быстрый переход к предыдущему контексту, `kubectx -`:

```
kubectx -
Switched to context "gke".
kubectx -
Switched to context "docker-for-desktop".
```

Если ввести просто `kubectx`, на экране появится список всех имеющихся у вас контекстов с выделением текущего.

Скорее всего, переключаться между пространствами имен вы будете чаще, чем между контекстами, и для этого идеально подходит утилита `kubens`:

```
kubens
default
kube-public
kube-system

kubens kube-system
Context "docker-for-desktop" modified.
Active namespace is "kube-system".

kubens -
Context "docker-for-desktop" modified.
Active namespace is "default".
```

Команды kubectx и kubens выполняют лишь одну функцию, но делают это хорошо. Они послужат отличным дополнением к инструментарию для работы с Kubernetes.

kube-ps1

Если вы используете командные оболочки `bash` или `zsh`, посоветуем для них небольшую утилиту (github.com/jonmosco/kube-ps1), которая добавляет текущий контекст Kubernetes в приглашение командной строки.

Установив `kube-ps1`, вы уже не сможете забыть, в каком контексте находитесь:

```
source "/usr/local/opt/kube-ps1/share/kube-ps1.sh"
PS1="[$(kube_ps1)]$ "
[(⎈ |docker-for-desktop:default)]
kubectx cloudnativedevops
Switched to context "cloudnativedevops".
(⎈ |cloudnativedevops:cloudnativedevopsblog)
```

Командные оболочки и инструменты Kubernetes

Обычной командной оболочки и `kubectl` вполне достаточно для большинства действий, которые вам нужно проделывать с кластером Kubernetes, но есть и другие варианты.

kube-shell

Если вам не хватает автозавершения `kubectl`, вы всегда можете воспользоваться оберткой для этой утилиты под названием `kube-shell`, которая предоставляет раскрывающийся список всех возможных вариантов для каждой команды (рис. 7.1).

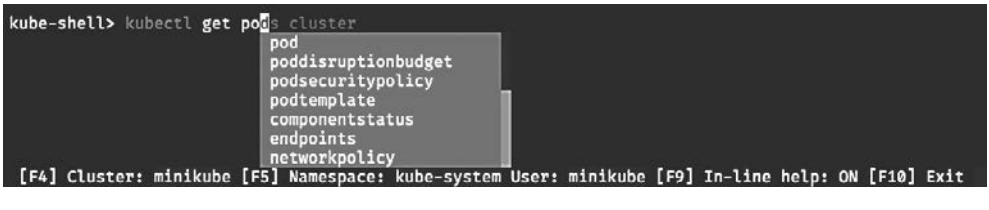

Рис. 7.1. kube-shell: интерактивный клиент для Kubernetes

Click

Click (databricks.com/blog/2018/03/27/introducing-click-the-command-line-interactive-controller-for-kubernetes.html) обеспечивает более гибкую работу с Kubernetes в терминале.

Click — это своего рода интерактивная версия `kubectl`, *запоминающая* текущий объект, с которым вы работаете. Например, если нужно найти и описать pod-оболочку с помощью `kubectl`, обычно сначала необходимо перечислить все подходящие pod-оболочки, затем скопировать и вставить в новую команду имя той из них, которая вас интересует.

Click вместо этого позволяет выбрать любой ресурс из списка по его номеру (например, `1` для первого элемента): он становится текущим и следующая команда Click будет работать с ним по умолчанию. Чтобы упростить поиск нужного вам объекта, Click поддерживает регулярные выражения.

Это мощный инструмент, который предоставляет очень приятную среду для работы с Kubernetes. Несмотря на то что Click считается *экспериментальным* и находится в *beta-версии*, он уже вполне подходит для выполнения ежедневных административных задач с кластером и его стоит опробовать.

kubed-sh

Если `kube-shell` и Click в сущности представляют собой локальные командные оболочки, которые немного знают о Kubernetes, то `kubed-sh` (произносится как *«кьюбд-аш»*) имеет более интригующую идею: это командная оболочка, которая в каком-то смысле выполняется *на* самом кластере.

`kubed-sh` загружает и запускает контейнеры, необходимые для работы в вашем кластере программ на JavaScript, Ruby или Python. Вы можете, к примеру, локально создать скрипт на Ruby и затем использовать `kubed-sh`, чтобы выполнить его в качестве развертывания Kubernetes.

Stern

Команда `kubectl logs` довольно полезна (см. подраздел «Просмотр журнальных записей контейнера» на с. 167), но не настолько удобна, как нам бы того хотелось. Например, перед ее использованием нужно выяснить уникальные имена pod-оболочки и контейнера, чьи журнальные записи вы хотите просмотреть, и затем указать их в командной строке — а это обычно требует как минимум одной операции копирования и вставки.

Кроме того, если вы используете флаг `--f` для слежения за записями конкретного контейнера, поток записей будет останавливаться всякий раз, когда этот контейнер перезапускается. Чтобы продолжить за ним следить, вам придется узнать его новое имя и снова выполнить команду `kubectl logs`. К тому же одновременно вы можете отслеживать журнальные записи только из одной pod-оболочки.

Более гибкий инструмент для поточного вывода журнала позволил бы нам указать группу pod-оболочек с помощью или регулярного выражения, которое соответствует их именам, или набора меток. Это позволило бы продолжать следить за журналами даже в случае перезапуска отдельных контейнеров.

К счастью, именно для этого и создан инструмент Stern (github.com/wercker/stern). Он выводит последние журнальные записи всех pod-оболочек, соответствующие регулярному выражению (например, `demo.*`). Если pod-оболочка содержит несколько контейнеров, Stern покажет вам сообщения от каждого, используя их имена в качестве префиксов.

Флаг `--since` позволяет ограничить вывод последними сообщениями (например, в пределах последних десяти минут).

Вместо сопоставления имен pod-оболочек с помощью регулярных выражений вы можете использовать любой селектор на основе меток Kubernetes, как и в случае с `kubectl`. В сочетании с флагом `--all-namespaces` это идеально подходит для наблюдения за журнальными записями нескольких контейнеров.

Создание собственных инструментов для работы с Kubernetes

В сочетании с такими инструментами для выполнения запросов, как `jq`, и стандартными утилитами Unix (`cut`, `grep`, `xargs` и т. д.) `kubectl` можно использовать для довольно сложной автоматизации работы с ресурсами Kubernetes. Как мы уже видели в этой главе, для написания скриптов также пригодится множество сторонних инструментов.

Однако у такого подхода есть свои ограничения. Это замечательно — приготовить изящный однострочный скрипт командной оболочки для интерактивной отладки и исследования, но он может оказаться слишком сложным для чтения и обслуживания.

Для настоящих системных программистов, которые автоматизируют промышленные рабочие процессы, мы настоятельно советуем использовать и настоящие

системные языки программирования. Язык Go — логичный выбор, поскольку он удовлетворил требованиям создателей Kubernetes, и, таким образом, данная платформа включает в себя полноценную клиентскую библиотеку (github.com/kubernetes/client-go) для программ на Go.

Библиотека `client-go` предоставляет полный доступ к API Kubernetes, поэтому вы можете делать с ней все то, что и с `kubectl`, и даже больше. Например, в следующем фрагменте кода показано, как вывести список всех pod-оболочек в вашем кластере:

```
...
podList, err := clientset.CoreV1().Pods("").List(metav1.ListOptions{})
if err != nil {
        log.Fatal(err)
}
fmt.Println("There are", len(podList.Items), "pods in the cluster:")
for _, i := range podList.Items {
        fmt.Println(i.ObjectMeta.Name)
}
...
```

Вы также можете создавать или удалять pod-оболочки, развертывания и любые другие ресурсы. Вы даже можете реализовать свой собственный тип ресурсов.

Если вам нужна функция, которой не хватает в Kubernetes, можете написать ее сами с помощью клиентской библиотеки.

У Kubernetes есть клиентские библиотеки (kubernetes.io/docs/reference/using-api/client-libraries) и для других языков, таких как Ruby, Python и PHP, их можно использовать аналогичным образом.

Резюме

У Kubernetes изумительное изобилие инструментов, и с каждой неделей их становится все больше. Вы, наверное, слегка подустали читать об очередной утилите, без которой, оказывается, нельзя обойтись, — и вас можно понять.

Но дело в том, что бóльшая часть этих инструментов вам не нужна. Платформа Kubernetes благодаря `kubectl` может сделать все, что потребуется. Остальное — лишь для развлечения и удобства.

Человек не может знать все на свете, но каждый из нас знает что-то. В эту главу мы включили советы и рекомендации от множества опытных Kubernetes-инженеров; в качестве источников использовались книги, статьи из блогов, документация

и парочка наших собственных открытий. Все, кому мы показывали этот материал, независимо от опыта смогли почерпнуть для себя кое-что полезное. И мы очень этому рады.

Вам стоит найти время и как следует познакомиться с утилитой `kubectl` и ее возможностями: для работы с Kubernetes это самый важный инструмент, который только есть, и вы будете очень часто им пользоваться.

Вот несколько самых важных моментов, которые следует знать.

- Утилита `kubectl` поставляется с исчерпывающей документацией о себе самой (`kubectl -h`) и обо всех ресурсах, полях и возможностях Kubernetes (`kubectl explain`).
- Если вам нужно выполнять сложные трансформации и фильтрацию с выводом `kubectl` (например, в скриптах), укажите формат JSON: `-o json`. Получив отформатированные данные, вы сможете выполнять к ним запросы, используя такие мощные инструменты, как `jq`.
- Вы можете передать `kubectl` параметр `--dry-run` и указать вывод в формате YAML с помощью `-o YAML`. Это позволит использовать императивные команды для генерации манифестов Kubernetes, а также сэкономит много времени при создании манифестов для новых приложений.
- Вы можете превратить в YAML-манифест и существующий ресурс, указав флаг `--export` для команды `kubectl get`.
- Команда `kubectl diff` показывает изменения, которые *могут* произойти в случае применения манифеста, ничего при этом не меняя.
- Вы можете просматривать вывод и сообщения об ошибках из любого контейнера с помощью команды `kubectl logs`; чтобы выводить их в виде потока, используйте флаг `--follow flag`, а для более гибкого наблюдения за журналами нескольких pod-оболочек подойдет Stern.
- Подключиться к проблемным контейнерам для их отладки можно с помощью команды `kubectl attach`; чтобы получить доступ к их командной оболочке, нужно выполнить `kubectl exec -it ... /bin/sh`.
- Команда `kubectl run` позволяет запустить любой образ контейнера. Например, для решения проблем можно воспользоваться универсальным инструментом BusyBox, который содержит все ваши любимые Unix-команды.
- Контексты Kubernetes работают по принципу закладок, помечая место в конкретном пространстве имен и кластере. Вы можете легко переключаться между контекстами и пространствами имен, используя инструменты `kubectx` и `kubens`.

- Click — это мощная командная оболочка для Kubernetes, которая предоставляет всю функциональность `kubectl` с сохранением состояния: она помнит текущий выбранный объект от команды к команде, чтобы вам не приходилось указывать его каждый раз.
- Платформа Kubernetes изначально поддерживает автоматизацию и управление из кода. Если вам нужно выйти за рамки того, что предоставляет `kubectl`, можете использовать клиентскую библиотеку `client-go`, которая дает полный контроль за всеми характеристиками кластера в коде на языке Go.

8 Работа с контейнерами

> Если есть сложный вопрос, на который вы не можете ответить, начните с более простого вопроса, на который не можете ответить.
>
> *Макс Тегмарк*

В предыдущих главах мы в основном уделяли внимание административным аспектам работы с Kubernetes: где размещать и как обслуживать ваши кластеры и как управлять их ресурсами. А теперь обратимся к наиболее фундаментальному объекту этой платформы — *контейнеру*. Мы рассмотрим, как работают контейнеры с технической точки зрения, как они связаны с pod-оболочками и как их образы развертывать в Kubernetes.

В этой главе мы также затронем важную тему безопасности контейнеров и покажем, как использовать функции Kubernetes для безопасного развертывания ваших приложений в соответствии с лучшими рекомендациями. В конце вы научитесь подключать дисковые тома к pod-оболочкам, позволяя контейнерам использовать и сохранять разделяемые данные.

Контейнеры и pod-оболочки

Мы уже познакомились с pod-оболочками в главе 2 и поговорили о том, как развертывания используют объект `ReplicaSet` для поддержания определенного количества копий pod-оболочек. Но при этом мы не останавливались подробно на самих объектах `Pod`. Pod-оболочка — это единица планирования в Kubernetes.

Pod-объект представляет собой контейнер или группу контейнеров, с его помощью в Kubernetes работают все остальные компоненты.

> Pod-оболочка — набор контейнеров с приложениями и томов, находящихся в единой среде выполнения. Наименьшим развертываемым объектом в кластере Kubernetes являются pod-оболочки, а не контейнеры. Это означает, что все контейнеры в pod-оболочке всегда оказываются на одном компьютере.
>
> *Келси Хайтауэр и др. Kubernetes Up & Running*

До сих пор термины *pod-оболочка* и *контейнер* в этой книге были в какой-то степени взаимозаменяемыми, так как pod-оболочка с демонстрационным приложением содержала лишь один контейнер. Однако в зависимости от сложности приложения один Pod-объект может состоять их двух и более контейнеров. Посмотрим, как это работает и зачем вам может понадобиться объединять контейнеры в pod-оболочки.

Что такое контейнер

Прежде чем спрашивать, зачем бы вам могло понадобиться несколько контейнеров в одной pod-оболочке, давайте вспомним, что же на самом деле представляет собой контейнер.

Из раздела «Пришествие контейнеров» на с. 34 вы знаете, что контейнер — это стандартизированный пакет, который содержит фрагмент программного обеспечения вместе с зависимостями контейнера, его конфигурацией, данными и т. д., то есть со всем, что нужно для выполнения. Но как это на самом деле работает?

В Linux и большинстве других операционных систем все, что выполняется на компьютере, заключено в *процессы*. Процесс представляет собой двоичный код и состояние в памяти запущенного приложения, такого как Chrome, iTunes или Visual Studio Code. Процессы находятся в едином глобальном пространстве имен: все они видят друг друга и могут взаимодействовать между собой; все используют один и тот же пул ресурсов, таких как процессор, память и файловая система (пространства имен в Linux и Kubernetes немного похожи, но, строго говоря, различны).

С точки зрения операционной системы контейнер представляет собой изолированный процесс (или группу процессов), который находится в своем собственном пространстве имен. Процессы внутри контейнера не видят ничего за его пределами, и наоборот. Контейнер не имеет доступа к ресурсам, принадлежащим другим

контейнерам или внешним процессам. Граница контейнера подобна ограждению, которое не дает процессам вырваться наружу и использовать ресурсы друг друга.

Что касается процесса внутри контейнера — он выполняется на отдельном компьютере с полным доступом ко всем его ресурсам и без каких-либо других процессов параллельно. В этом можно убедиться, если запустить в контейнере несколько команд:

```
kubectl run busybox --image busybox:1.28 --rm --it --restart=Never /bin/sh
If you don't see a command prompt, try pressing enter.
/ # ps ax
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
    8 root      0:00 ps ax

/ # hostname
busybox
```

Обычно команда `ps ax` выводит список всех процессов, запущенных на компьютере, и их, как правило, довольно много (несколько сотен на типичном Linux-сервере). Но мы видим здесь лишь два процесса: `/bin/sh` и `ps ax`. Таким образом, в контейнере видны только те процессы, которые в нем выполняются.

Точно так же команда `hostname`, которая в нормальных условиях показала бы сетевое имя компьютера, возвращает `busybox`: на самом деле это имя контейнера. Итак, с точки зрения контейнера `busybox` все выглядит так, как будто он работает на компьютере с именем `busybox` и система принадлежит только ему. Это справедливо для всех контейнеров, запущенных на том же компьютере.

В качестве интересного упражнения можете сами создать контейнер, не используя такие среды выполнения, как Docker. В отличном докладе Лиз Райс под названием «Что же такое контейнер на самом деле?» (youtu.be/HPuvDm8IC-4) показано, как это делается с нуля в программе на Go.

Что должно находиться в контейнере

Не существует технических ограничений относительно количества процессов, которые можно запускать внутри контейнера: вы можете поместить в него целый дистрибутив Linux с несколькими активными приложениями, сетевыми сервисами и т. д. В связи с этим контейнеры иногда называют *легковесными виртуальными машинами*. Но это не самый лучший способ их использования, так как вы теряете все преимущества от изоляции процессов.

Если процессам не нужно знать друг о друге, значит, их не обязательно размещать в одном контейнере. Практика показывает, что лучше поручить контейнеру *какую-то одну функцию*. Например, контейнер нашего демонстрационного приложения прослушивает сетевой порт и шлет строку `Hello, 世界` любому, кто к нему подключится. Это простой, самодостаточный сервис: он не зависит ни от каких других программ или сервисов и при этом ничто не зависит от него — идеальный кандидат на получение отдельного контейнера.

У контейнера также есть *точка входа* — команда, которая запускается при старте. Обычно она приводит к созданию единого процесса для выполнения команды, хотя какие-то приложения запускают несколько вспомогательных или рабочих подпроцессов. Для запуска больше одного отдельного процесса в контейнере нужно написать оберточный скрипт, который будет играть роль входной точки, запуская нужные вам процессы.

Каждый контейнер должен выполнять лишь один главный процесс. Если вы запускаете большое количество не связанных между собой процессов, то не в полной мере используете мощь контейнера и вам следует подумать о разбивке вашего приложения на несколько взаимодействующих между собой контейнеров.

Что должно находиться в pod-оболочке

Итак, теперь вы знаете, что такое контейнеры, и понимаете, почему их полезно объединять в pod-оболочки. `Pod`-объект представляет собой группу контейнеров, которые должны взаимодействовать и обмениваться данными; планировать, запускать и останавливать их нужно вместе, и находиться они должны на одном и том же физическом компьютере.

Хорошим примером может служить приложение, которое хранит данные в локальном кэше, таком как Memcached (memcached.org/about). Вам придется запускать два процесса: ваше приложение и сервер `memcached`, отвечающий за хранение и извлечение данных. И хотя оба процесса можно поместить в единый контейнер, делать это не обязательно: им для взаимодействия достаточно сетевого сокета. Лучше разделить их на два отдельных контейнера, каждый из которых должен заботиться о сборке и выполнении только своего собственного процесса.

На самом деле вы можете использовать публичный образ контейнера Memcached, доступный на Docker Hub: он уже готов к работе с другим контейнером в рамках одной pod-оболочки.

Итак, вы создаете pod-оболочку с двумя контейнерами: Memcached и своим приложением. Приложение может общаться с Memcached, соединяясь по сети. Поскольку оба контейнера находятся в одной pod-оболочке, а следовательно, и на одном узле, это соединение всегда будет локальным.

Точно так же можно представить себе блог-приложение, состоящее из двух контейнеров: веб-сервера, такого как Nginx, и программы для синхронизации, которая клонирует репозиторий Git с данными блога (HTML-файлами, изображениями и т. д.). Блог записывает данные на диск, и поскольку pod-оболочка предоставляет общий дисковый том, они будут доступны и для контейнера с Nginx, который сможет раздавать их по HTTP.

> В целом при проектировании pod-оболочек следует задаться вопросом: «Будут ли эти контейнеры работать корректно, если окажутся на разных компьютерах?» Если ответ отрицательный, контейнеры сгруппированы верно. Если ответ положительный, правильным решением, скорее всего, будет разделить их на несколько pod-оболочек.
>
> *Келси Хайтауэр и др. Kubernetes Up & Running*

Все контейнеры в pod-оболочке должны работать вместе над выполнением одной задачи. Если вам для этого нужен лишь один контейнер, отлично — ограничьтесь одним контейнером. Если нужно два или три — тоже не проблема. Но если контейнеров получается еще больше, наверное, следует подумать о том, чтобы разбить их по отдельным pod-оболочкам.

Манифесты контейнеров

Мы объяснили, что такое контейнеры, что в них должно содержаться и когда их нужно объединять в pod-оболочки. Теперь посмотрим, как же запустить контейнер в Kubernetes.

Когда в подразделе «Манифесты развертываний» на с. 96 вы создали свое первое развертывание, оно содержало раздел `template.spec`, в котором был указан контейнер для запуска (в том примере был всего один контейнер):

```
spec:
  containers:
  - name: demo
    image: cloudnatived/demo:hello
    ports:
    - containerPort: 8888
```

Ниже показано, как бы выглядел раздел развертывания `template.spec` с двумя контейнерами:

```
spec:
  containers:
  - name: container1
    image: example/container1
  - name: container2
    image: example/container2
```

Единственными обязательными полями в спецификации каждого контейнера являются `name` (имя должно быть, чтобы на него могли ссылаться другие ресурсы) и `image` (вы должны сказать Kubernetes, какой образ следует запустить в этом контейнере).

Идентификаторы образов

Вы уже использовали в этой книге идентификаторы разных образов контейнеров: например, `cloudnatived/demo:hello`, `alpine` и `busybox:1.28`.

На самом деле идентификатор образа состоит из четырех разных частей: *сетевого имени реестра, пространства имен репозитория, репозитория образа* и *тега*. Обязательным является лишь имя образа. Идентификатор с использованием всех этих элементов выглядит так:

```
docker.io/cloudnatived/demo:hello
```

- Сетевым именем реестра в данном примере является `docker.io`; на самом деле это значение по умолчанию для образов Docker, поэтому указывать его не обязательно. Хотя, если ваш образ хранится в другом реестре, вы должны указать его сетевое имя. Например, образы из Google Container Registry имеют префикс `gcr.io`.
- Пространство имен репозитория равно `cloudnatived`: это мы (привет!). Если его не указать, будет использовано значение по умолчанию (`library`). Вот список официальных образов (docs.docker.com/docker-hub/official_images), поддерживаемых компанией Docker Inc. Среди них особой популярностью пользуются базовые образы ОС (`alpine`, `ubuntu`, `debian`, `centos`), языковые среды (`golang`, `python`, `ruby`, `php`, `java`) и широко используемое программное обеспечение (`mongo`, `mysql`, `nginx`, `redis`).
- Репозиторий образа называется `demo` и относится к конкретному образу контейнера внутри реестра и пространства имен (см. также подраздел «Контрольные суммы контейнеров» на с. 190).

- В качестве тега указано `hello`. Теги определяют разные версии одного и того же образа.

Выбор тегов для контейнера зависит лишь от вас. Есть несколько популярных вариантов:

- теги с семантическими версиями, такие как `v1.3.0`. Обычно указывают на версию приложения;
- тег на основе Git SHA вида `5ba6bfd...`. Указывает на определенную фиксацию в репозитории исходного кода, которая использовалась при сборке контейнера (см. подраздел «Теги на основе Git SHA» на с. 320);
- тег, представляющий среду, такую как `staging` или `production`.

Вы можете добавить в свой образ столько тегов, сколько захотите.

Тег latest

Если тег не указать, при загрузке образа по умолчанию будет применяться значение `latest`. Например, когда вы используете образ `alpine` без каких-либо тегов, вы получаете `alpine:latest`.

Тег `latest` добавляется по умолчанию, когда вы собираете или загружаете образ в репозиторий без указания других тегов. Он ссылается не на последнюю версию образа, а на ту, которая последней не была явно помечена тегами. Из-за этого данный тег не очень полезен (vsupalov.com/docker-latest-tag) в качестве идентификатора.

Поэтому при развертывании промышленных контейнеров в Kubernetes важно всегда использовать определенный тег. Если требуется быстро запустить одноразовый контейнер (например, `alpine`) для отладки или экспериментов, тегами можно пренебречь, чтобы получить последнюю версию образа. Но если вы имеете дело с настоящими приложениями, нужно быть уверенными в том, что при развертывании pod-оболочек вы каждый раз получаете одни и те же образы контейнеров.

> При развертывании контейнеров в промышленной среде следует избегать тега latest, поскольку это затрудняет отслеживание версий текущих образов и их откат.
>
> *Документация Kubernetes*[1]

[1] kubernetes.io/docs/concepts/configuration/overview/#using-labels.

Контрольные суммы контейнеров

Как мы уже видели, тег `latest` не всегда означает то, что мы думаем, но даже теги на основе семантических версий или Git SHA не могут однозначно и окончательно идентифицировать конкретный образ контейнера. Если мейнтейнер решит загрузить в репозиторий другой образ с тем же тегом, при вашем следующем развертывании вы получите его обновленную версию. Говоря техническим языком, теги являются *недетерминистическими*.

Иногда желательно иметь детерминистические развертывания, то есть гарантировать, что развертывание всегда ссылается именно на тот образ контейнера, который вы указали. Этого можно добиться с помощью *контрольной суммы* контейнера: криптографического хеша, который неизменно идентифицирует данный образ.

Образы могут иметь много тегов, но только одну контрольную сумму. Это означает, что при задании в манифесте контейнера контрольной суммы образа вы гарантируете детерминистические развертывания. Идентификатор образа с контрольной суммой выглядит так:

```
cloudnatived/
demo@sha256:aeae1e551a6cbd60bcfd56c3b4ffec732c45b8012b7cb758c6c4a34...
```

Теги базового образа

Ссылаясь на базовый образ в Dockerfile, вы получите его версию `latest`, если не укажете конкретный тег, — точно так же, как это происходит при развертывании контейнера. Ввиду запутанной семантики `latest`, о которой мы говорили выше, для базовых образов лучше использовать определенные теги, такие как `alpine:3.8`.

Когда вы редактируете код приложения и пересобираете его контейнер, вам вряд ли захочется получить неожиданные изменения из-за более нового публичного базового образа. Это может вызвать проблемы, которые будет сложно найти и отладить.

Чтобы сделать свои сборки как можно более воспроизводимыми, используйте конкретный тег или контрольную сумму.

Мы советовали вам избегать тега latest, однако следует отметить, что на этот счет существуют разные мнения и даже у авторов данной книги разные предпочтения. Если всегда использовать последние базовые образы, вы сразу же сможете обнаруживать проблемы, которые ломают вашу сборку. Задание конкретных тегов также означает, что вы будете обновлять свой базовый образ только тогда, когда сами этого захотите, независимо от мейнтейнеров исходного образа. Решать вам.

Порты

Вы уже видели, как мы использовали поле `ports` в нашем демонстрационном приложении: в нем перечислялись номера сетевых портов, которые будет прослушивать приложение. Это поле имеет описательный характер и не имеет особого значения для Kubernetes, но его рекомендуется указывать.

Запросы и лимиты на ресурсы

Мы уже подробно обсуждали запросы и лимиты на ресурсы в главе 5, поэтому лишь кратко напомним, о чем идет речь.

Каждый контейнер может предоставить один или несколько параметров в своей спецификации:

- `resources.requests.cpu`;
- `resources.requests.memory`;
- `resources.limits.cpu`;
- `resources.limits.memory`.

Запросы и лимиты указываются для отдельных контейнеров, но обычно мы говорим о них в контексте ресурсов pod-оболочки. В таком случае запрос ресурсов будет суммой всех запросов для всех контейнеров этой pod-оболочки и т. д.

Политика загрузки образов

Как вы знаете, прежде чем запустить контейнер на узле, его образ необходимо *загрузить* или скачать из соответствующего реестра. Поле `imagePullPolicy` в спецификации контейнера определяет, насколько часто Kubernetes будет это делать. Возможно одно из трех значений: `Always`, `IfNotPresent` или `Never`.

- `Always`. Образ будет загружаться при каждом запуске контейнера. Если предположить, что вы указали тег (как это сделать, см. подраздел «Тег latest» на с. 189), такая политика является излишней и впустую тратит время и трафик.
- `IfNotPresent`. Используется по умолчанию и подходит для большинства ситуаций. Если образ еще не присутствует на узле, он загрузится и после этого будет использоваться при каждом запуске контейнера, пока вы не измените его спецификацию. Kubernetes не станет пытаться загрузить его заново.
- `Never`. Образ никогда не обновляется. С такой политикой Kubernetes никогда не станет загружать образ из реестра, будет использоваться тот образ, который уже присутствует на узле. Если его нет, контейнер не сможет запуститься. Вряд ли вам это подойдет.

Если вы столкнетесь со странными проблемами (например, pod-оболочка не обновляется после загрузки в репозиторий нового образа контейнера), проверьте политику загрузки образов.

Переменные среды

Использование переменных среды — это распространенный, хотя и ограниченный способ передачи информации контейнеру на этапе выполнения. Он распространен ввиду того, что доступ к этим переменным имеют все исполняемые файлы в Linux, и даже программы, написанные задолго до появления контейнеров, могут использовать их для конфигурации. Ограниченность способа связана с тем, что переменные среды могут иметь лишь строковые значения: никаких массивов, словарей или любых других структурированных данных. Кроме того, общий размер среды процесса не может превышать 32 КиБ, поэтому невозможно передавать через него большие файлы.

Чтобы установить переменную среды, укажите ее в поле `env` своего контейнера:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  env:
  - name: GREETING
    value: "Hello from the environment"
```

Если образ контейнера сам указывает переменные среды (например, в Dockerfile), параметр `env`, принадлежащий Kubernetes, их перезапишет. Это может пригодиться для изменения конфигурации контейнера по умолчанию.

Более гибкий способ передачи конфигурационных данных в контейнеры состоит в использовании таких объектов Kubernetes, как ConfigMap или Secret (подробнее об этом — в главе 10).

Безопасность контейнеров

При выводе списка процессов в контейнере с помощью команды `ps ax` (см. подраздел «Что такое контейнер» на с. 184) вы могли заметить, что все процессы выполнялись от имени пользователя `root`. В Linux и других Unix-подобных операционных системах `root` является администратором и имеет право на чтение любых данных, изменение любого файла и выполнение любой операции в системе.

В полноценных системах Linux некоторые процессы (например, `init`, который управляет всеми остальными процессами) должны работать от имени администратора, но в контейнерах обычно это не так.

Действительно, запускать процессы от имени пользователя `root`, когда этого не требуется, — плохая идея. Она противоречит принципу *минимальных привилегий* (ru.wikipedia.org/wiki/Принцип_минимальных_привилегий), согласно которому программа должна иметь доступ только к действительно необходимым ей для работы данным и ресурсам.

Для любого, кто писал программы, очевиден тот факт, что в них бывают ошибки. Некоторые ошибки позволяют злоумышленникам взломать программу и заставить ее делать то, чего она делать не должна: например, прочитать секретные данные или выполнить произвольный код. Чтобы сократить подобные риски, контейнеры следует запускать с минимально возможными привилегиями.

Начнем с того, что они должны работать от имени *обычного* пользователя (не имеющего особых привилегий, таких как чтение файлов других пользователей), а не администратора.

> На своем сервере вы ничего не запускаете от имени администратора (по крайней мере, это нежелательно). Точно так же ничто в вашем контейнере не должно работать с администраторскими привилегиями. Запуск двоичных файлов, созданных в других местах, требует значительного доверия. Это относится и к двоичным файлам внутри контейнеров.
>
> *Марк Кэмпбелл*[1]

Кроме того, взломщики могут воспользоваться ошибкой в среде выполнения контейнера, чтобы выйти за его пределы и получить те же права и привилегии на основном компьютере.

Запуск контейнеров от имени обычного пользователя

Ниже показан пример спецификации контейнера, которая заставляет Kubernetes запустить контейнер от имени определенного пользователя:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    runAsUser: 1000
```

[1] medium.com/@mccode/processes-in-containers-should-not-run-as-root-2feae3f0df3b.

В поле `runAsUser` указан *UID* (numerical user identifier — числовой идентификатор пользователя). Во многих дистрибутивах Linux UID 1000 присваивается первому обычному пользователю, который создается в системе, поэтому для UID-контейнеров обычно лучше выбирать значения от 1000 и выше. Неважно, *существует* ли в контейнере пользователь с таким идентификатором; это будет работать даже для контейнеров, созданных с нуля, у которых нет операционной системы.

Docker также позволяет указать в Dockerfile пользователя, от имени которого будут выполняться процессы контейнера, но вам это делать не надо: задание поля `runAsUser` в спецификации Kubernetes является более простым и гибким способом.

Идентификатор, указанный в `runAsUser`, переопределяет пользователя, сконфигурированного в образе контейнера. Если поле `runAsUser` проигнорировано, а пользователь указан в самом контейнере, Kubernetes будет использовать именно его. Если же пользователь не определен ни в манифесте, ни в образе, контейнер будет работать от имени `root` (что, как мы уже написали, плохая идея).

Для максимальной безопасности каждый контейнер должен иметь отдельный UID. Например, если он каким-то образом будет скомпрометирован или случайно попытается перезаписать данные, он не навредит другим контейнерам, потому что имеет доступ только к своим собственным данным.

Но если нужно, чтобы два контейнера или более могли работать с одними и теми же данными (например, через подключенный том), вы должны назначить им одинаковые UID.

Блокирование контейнеров с администраторскими привилегиями

Чтобы предотвратить такую ситуацию, Kubernetes позволяет блокировать запуск контейнеров, которые собираются работать от имени администратора.

Для этого предусмотрен параметр `runAsNonRoot: true`:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    runAsNonRoot: true
```

Когда контейнер попытается запуститься, Kubernetes проверит, будет ли тот работать от имени администратора, и в случае положительного результата откажет в запуске. Так что у вас не будет проблем, даже если вы забудете указать обычного

пользователя для своих контейнеров или если используете сторонние контейнеры, которые сконфигурированы для работы от имени `root`.

В этой ситуации pod-оболочка будет иметь состояние `CreateContainerConfigError`, и, выполнив для нее команду `kubectl describe`, можно будет увидеть такую ошибку:

```
Error: container has runAsNonRoot and image will run as root
```

Рекомендуемый подход

Запускайте контейнеры от имени обычных пользователей и блокируйте запуск контейнеров с администраторскими привилегиями с помощью параметра runAsNonRoot: true.

Настройка файловой системы только для чтения

Еще одна настройка, полезная с точки зрения безопасности, — `readOnlyRootFilesystem`, не позволяющая контейнеру записывать в его собственную файловую систему. Вполне возможно представить ситуацию, когда контейнер пользуется уязвимостью в Docker или Kubernetes, вследствие которой могут измениться файлы на родительском узле через запись в локальную файловую систему. Если файловая система находится в режиме «только для чтения», этого не произойдет, потому что контейнер получит ошибку ввода/вывода:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    readOnlyRootFilesystem: true
```

Многим контейнерам не нужно ничего записывать в собственную файловую систему, поэтому подобная конфигурация не помешает их работе. Параметр `readOnlyRootFilesystem` рекомендуется (kubernetes.io/blog/2016/08/security-best-practices-kubernetes-deployment) устанавливать всегда, если только контейнеру действительно не нужно производить запись в файлы.

Отключение повышения привилегий

Обычно исполняемые файлы в Linux работают с теми же привилегиями, что и пользователь, их запускающий. Но есть исключение: файлы, использующие механизм `setuid`, могут временно получить привилегии пользователя, который ими *владеет* (обычно это `root`).

Для контейнера это является потенциальной проблемой: даже если он работает от имени обычного пользователя (с UID 1000, например), но в нем есть исполняемый файл с `setuid`, этот файл может по умолчанию получить администраторские привилегии.

Чтобы подобного не случилось, присвойте полю `allowPrivilegeEscalation` в политике безопасности контейнера значение `false`:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    allowPrivilegeEscalation: false
```

Вы можете управлять этой настройкой на уровне всего кластера, а не отдельного контейнера (см. подраздел «Политики безопасности pod-оболочек» на с. 198).

Современным программам в Linux не нужен бит `setuid`, они могут достичь того же результата с помощью более гибкого механизма привилегий под названием *«мандаты»*.

Мандаты

Традиционно программы в Unix имеют два уровня привилегий: *обычный* и *администраторский*. Если у обычных программ привилегий не больше, чем у пользователей, которые их запускают, то администраторские программы могут делать что угодно, обходя любые проверки безопасности со стороны ядра.

Механизм мандатов в Linux является шагом вперед. Он четко определяет, что программа может делать: загружать модули ядра, напрямую выполнять сетевые операции ввода/вывода, обращаться к системным устройствам и т. д. Если программе нужна определенная привилегия, она может получить именно ее и никакие другие.

Например, веб-сервер, который прослушивает порт 80, должен выполняться от имени администратора: номера портов ниже 1024 считаются системными и привилегированными. Вместо этого программе может быть выдан мандат `NET_BIND_SERVICE`, который позволит привязать ее к любому порту, но не даст никаких других особых привилегий.

Контейнеры Docker по умолчанию имеют довольно богатый набор мандатов. Это прагматичное решение, основанное на компромиссе между безопасностью и удобством: если контейнерам изначально не выдавать *никаких* мандатов, этим каждый раз придется заниматься самому пользователю, который их запускает.

С другой стороны, в соответствии с принципом минимальных привилегий контейнер не должен иметь ненужных ему мандатов. Контекст безопасности Kubernetes

позволяет убрать любой мандат из стандартного набора или добавить его туда в случае необходимости. Например:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    capabilities:
      drop: ["CHOWN", "NET_RAW", "SETPCAP"]
      add: ["NET_ADMIN"]
```

Контейнер будет лишен мандатов `CHOWN`, `NET_RAW` и `SETPCAP`, но получит мандат `NET_ADMIN`.

В документации для Docker (docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities) перечислены все мандаты, которые назначаются контейнерам по умолчанию, и все, которые при необходимости можно добавить.

Для максимальной безопасности следует убрать любые мандаты для контейнеров и затем по мере необходимости отдельно их выдавать:

```
containers:
- name: demo
  image: cloudnatived/demo:hello
  securityContext:
    capabilities:
      drop: ["all"]
      add: ["NET_BIND_SERVICE"]
```

Механизм мандатов жестко ограничивает то, что могут делать процессы внутри контейнеров, даже если они запущены от имени `root`. Если мандат был убран на уровне контейнера, его не сможет выдать даже вредоносный процесс с максимальными привилегиями.

Контексты безопасности pod-оболочки

Мы уже обсуждали параметры контекста безопасности на уровне отдельных контейнеров. Но некоторые из них можно устанавливать и на уровне pod-оболочки:

```
apiVersion: v1
kind: Pod
...
  spec:
    securityContext:
    runAsUser: 1000
    runAsNonRoot: false
    allowPrivilegeEscalation: false
```

Эти настройки будут действовать для всех контейнеров в pod-оболочке, если только контейнер не перезапишет данный параметр в своем собственном контексте безопасности.

Рекомендуемый подход

Устанавливайте контексты безопасности для всех своих pod-оболочек и контейнеров. Отключайте повышение привилегий и убирайте все мандаты. Выдавайте только те мандаты, которые необходимы вашему контейнеру.

Политики безопасности pod-оболочки

Вместо того чтобы указывать параметры безопасности для каждой отдельной pod-оболочки или контейнера, вы можете описать их на уровне кластера с помощью ресурса `PodSecurityPolicy`, который выглядит следующим образом:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: false
  # Дальше идут некоторые обязательные поля.
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
  - *
```

Эта простая политика блокирует привилегированные контейнеры, то есть те, у которых установлен флаг `privileged` в их контексте безопасности и которые в результате получили бы всю свободу процесса, запущенного прямо на узле.

`PodSecurityPolicie` использовать не так и просто, поскольку вам придется самим создавать политики, выдавать им доступ к нужным служебным учетным записям через RBAC (см. подраздел «Введение в управление доступом на основе ролей» на с. 258) и включать контроллер доступа `PodSecurityPolicy` в своем кластере. Но если у вас большая инфраструктура или вы не контролируете напрямую

конфигурацию безопасности отдельных pod-оболочек, `PodSecurityPolicy` будет хорошим выбором.

О том, как создавать и включать политики безопасности pod-оболочек, можно почитать в документации Kubernetes.

Служебные учетные записи pod-оболочек

Если ничего не менять (см. подраздел «Приложения и развертывание» на с. 262), pod-оболочки будут работать с правами доступа служебной учетной записи, которая по умолчанию действует в заданном пространстве имен. Если же по каким-то причинам нужно будет выдать дополнительные права (например, доступ к pod-оболочкам из других пространств имен), можно создать для приложения отдельную служебную учетную запись, привязать ее к необходимым ролям и прописать в конфигурации pod-оболочки.

Для этого укажите в поле `serviceAccountName` спецификации pod-оболочки имя служебной учетной записи:

```
apiVersion: v1
kind: Pod
...
spec:
  serviceAccountName: deploy-tool
```

Тома

Как вы помните, у каждого контейнера есть своя файловая система, доступная только ему. Она *временная*: любые данные, не входящие в состав образа контейнера, теряются при его перезапуске.

Обычно это неплохо: демонстрационное приложение, к примеру, является сервером, который не хранит состояние, поэтому ему не требуется постоянное хранилище. Ему также не нужно делиться файлами с другими контейнерами.

Однако более сложным приложениям может понадобиться возможность разделять данные на совместное использование другими контейнерами в одной pod-оболочке и сохранять их между перезапусками. Эту возможность предоставляет объект `Volume` (том) из состава Kubernetes.

Есть много разных типов томов, которые можно подключить к pod-оболочке. Какой бы носитель информации ни использовался внутри, подключенный

к pod-оболочке том доступен всем ее контейнерам. Если контейнерам нужно общаться посредством обмена файлами, делать это они могут с помощью томов того или иного вида. Самые важные типы томов будут рассмотрены в следующих разделах.

Тома emptyDir

Самый простой тип томов называется `emptyDir`. Это экземпляр временного хранилища, которое создается пустым (отсюда и название) и размещает свои данные на узле (либо в оперативной памяти, либо на диске). Оно существует, пока на этом узле находится его pod-оболочка.

Том `emptyDir` подходит для случаев, когда нужно выделить для контейнера немного дополнительного места, но при этом вы не возражаете, если данные в какой-то момент пропадут или переместятся на другой узел вместе с контейнером. В качестве примера можно привести кэширование загруженных файлов и сгенерированных данных или использование минимальной рабочей среды для обработки информации.

Аналогично, если вы хотите обмениваться файлами между контейнерами в pod-оболочке, но вам не нужно хранить эти данные на протяжении долгого времени, идеально подойдет том `emptyDir`.

Вот пример pod-оболочки, которая создает том `emptyDir` и подключает его к контейнеру:

```
apiVersion: v1
kind: Pod
...
spec:
  volumes:
  - name: cache-volume
    emptyDir: {}
  containers:
  - name: demo
    image: cloudnatived/demo:hello
  volumeMounts:
  - mountPath: /cache
    name: cache-volume
```

Для начала мы создаем том `emptyDir` под названием `cache-volume` в разделе `volumes` спецификации pod-оболочки:

```
volumes:
- name: cache-volume
  emptyDir: {}
```

Теперь том `cache-volume` доступен для подключения и использования в любом контейнере данного `Pod`-объекта. Для этого его нужно указать в разделе `volumeMounts` контейнера `demo`:

```
name: demo
image: cloudnatived/demo:hello
volumeMounts:
- mountPath: /cache
  name: cache-volume
```

Чтобы использовать новое хранилище, контейнеру не нужно делать ничего особенного: все, что он записывает в каталог `/cache`, будет попадать в том и становиться доступным для других контейнеров, к которым этот том также подключен. Все контейнеры, подключившие том, могут производить с ним операции чтения и записи.

Будьте осторожны при записи в общие тома: Kubernetes не блокирует запись на диске. Если два контейнера попытаются одновременно записать в один и тот же файл, данные могут быть повреждены. Чтобы этого избежать, вам следует либо реализовать собственный механизм блокировки записи, либо использовать том другого типа с поддержкой блокировок, такой как glusterfs.

Постоянные тома

Том `emptyDir` идеально подходит для кэширования и обмена временными файлами, однако некоторым приложениям — например, базам данных — необходимо хранить постоянную информацию. В целом мы не советуем запускать базы данных в Kubernetes: для этого почти всегда лучше использовать облачный сервис. Например, большинство облачных провайдеров предлагают управляемые решения для реляционных баз данных вроде MySQL и PostgreSQL, а также для хранилищ типа «ключ — значение» (NoSQL).

Как уже говорилось в подразделе «Kubernetes не решает все проблемы» на с. 42, данная платформа лучше всего подходит для работы с приложениями, которые не хранят состояние (то есть не требуют постоянных данных). Постоянное хранение данных значительно усложняет конфигурацию Kubernetes для вашего приложения, потому что использует дополнительные облачные ресурсы и требует наличия резервного копирования.

Но если вам в Kubernetes нужны постоянные тома, ресурс `PersistentVolume` — это то, что вы ищете. Не станем углубляться в подробности, так как они могут различаться в зависимости от вашего облачного провайдера: больше о `PersistentVolume` можно почитать в документации Kubernetes (kubernetes.io/docs/concepts/storage/persistent-volumes).

Самый гибкий способ использования постоянных томов в Kubernetes состоит в создании объекта `PersistentVolumeClaim`. Это запрос тома `PersistentVolume` определенного типа и размера: например, тома объемом 10 ГиБ с высокоскоростным хранилищем, доступным для чтения и записи.

Pod-оболочка в итоге может добавить этот запрос `PersistentVolumeClaim` в качестве тома, и ее контейнеры смогут его подключить и использовать:

```
volumes:
- name: data-volume
  persistentVolumeClaim:
    claimName: data-pvc
```

Вы можете создать в своем кластере пул постоянных томов, которые pod-оболочки будут запрашивать вышеописанным образом. В качестве альтернативы можно настроить *динамическое выделение* (kubernetes.io/docs/concepts/storage/dynamic-provisioning): когда подключается подобный запрос PersistentVolumeClaim, выделяется соответствующая часть хранилища, которая затем подключается к `Pod`-объекту.

Политики перезапуска

Из подраздела «Запуск контейнеров с целью отладки» на с. 170 вы узнали, что Kubernetes всегда перезапускает pod-оболочки, которые завершают работу (если только вы не измените эту настройку). Таким образом, по умолчанию используется политика перезапуска `Always`, но вы можете указать вместо нее `OnFailure` (перезапускать, только если контейнер завершил работу с ненулевым статусом) или `Never`:

```
apiVersion: v1
kind: Pod
...
spec:
  restartPolicy: OnFailure
```

Если вы не хотите, чтобы ваша pod-оболочка перезапускалась после завершения работы, используйте ресурс `Job` (см. подраздел «Запланированные задания» на с. 220).

imagePullSecrets

Как вы уже знаете, если образ не присутствует на узле, Kubernetes загрузит его из реестра контейнеров. Но что, если вы используете приватный реестр? Как передать Kubernetes учетные данные для аутентификации в таком реестре?

Сделать это можно с помощью поля `imagePullSecrets` в pod-оболочке. Для начала учетные данные необходимо сохранить в объекте `Secret` (подробнее об этом — в разделе «Конфиденциальные данные в Kubernetes» на с. 243), после чего можно указать этот объект, чтобы он использовался при загрузке любых контейнеров в pod-оболочке. Например, если объект `Secret` называется `registry-creds`:

```
apiVersion: v1
kind: Pod
...
spec:
imagePullSecrets:
- name: registry-creds
```

Формат учетных данных для реестра описывается в документации Kubernetes (kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry).

`imagePullSecrets` можно также привязать к служебной учетной записи (см. подраздел «Служебные учетные записи pod-оболочек» на с. 199). Любая pod-оболочка, созданная такой учетной записью, автоматически получит доступ к привязанным учетным данным реестра.

Резюме

Чтобы понять Kubernetes, сначала нужно понять контейнеры. В этой главе мы дали вам представление о них и о том, как они работают вместе с pod-оболочками и какие опции для управления контейнерами в Kubernetes вам доступны.

Самое важное.

- На уровне ядра Linux контейнер представляет собой изолированный набор процессов с обособленными ресурсами. С точки зрения контейнера все выглядит так, как будто ему выделен целый компьютер с Linux.
- Контейнеры не являются виртуальными машинами. Каждый из них должен выполнять лишь один основной процесс.
- Pod-оболочка обычно содержит один контейнер с основным приложением и опциональный *вспомогательный* контейнер для поддержки.
- Полное название образа контейнера может включать в себя сетевое имя реестра, пространство имен репозитория, репозиторий образа и тег: например, `docker.io/cloudnatived/demo:hello`. Обязательным является только имя образа.
- Чтобы ваши развертывания можно было воспроизвести, всегда указывайте тег для образа контейнера. В противном случае вы получите последнюю текущую версию.

- Программы в контейнерах не должны выполняться от имени администратора. Вместо этого для них следует назначить обычного пользователя.
- Вы можете указать поле `runAsNonRoot: true`, чтобы блокировать любые контейнеры, которые хотят работать от имени `root`.
- Среди прочих полезных параметров безопасности контейнеров можно выделить `readOnlyRootFilesystem: true` и `allowPrivilegeEscalation: false`.
- Мандаты Linux предоставляют гибкий механизм управления привилегиями, но контейнеры по умолчанию имеют слишком щедрый их набор. Для начала уберите все мандаты, а затем выдавайте контейнеру необходимые.
- Контейнеры в одной pod-оболочке могут обмениваться данными путем чтения и записи в подключенном томе. Самый простой тип томов, `emptyDir`, изначально является пустым и хранит свое содержимое только во время работы pod-оболочки.
- Том `PersistentVolume`, в свою очередь, хранит свое содержимое столько, сколько нужно. Pod-оболочки могут динамически выделять новые постоянные тома с помощью запросов `PersistentVolumeClaim`.

9 Управление pod-оболочками

> Нет больших проблем, есть просто множество маленьких.
>
> *Генри Форд*

В предыдущей главе мы поговорили о некоторых деталях работы контейнеров и рассказали, как они складываются в pod-оболочки. У pod-оболочек есть несколько других интересных аспектов, к которым мы обратимся в этой главе: например, метки, управление планированием за счет принадлежности к узлам, блокирование запуска на определенных узлах с помощью ограничений и допусков, запуск вместе или по отдельности с использованием принадлежности pod-оболочек и оркестрация приложений посредством контроллеров pod-оболочек вроде `DaemonSets` и `StatefulSets`.

Мы также расскажем о некоторых расширенных сетевых возможностях, включая ресурсы Ingress, Istio и Envoy.

Метки

Как вы знаете, pod-оболочкам (и другим ресурсам Kubernetes) можно назначать метки. Они играют важную роль в объединении связанных между собой ресурсов (например, когда вы направляете запросы от сервиса к соответствующим внутренним компонентам). В этом разделе мы детально рассмотрим метки и селекторы.

Что такое метки

> Метки — это пары типа «ключ — значение», которые назначаются объектам вроде pod-оболочек. Они предназначены для определения значимых для пользователей идентификационных атрибутов объектов, но при этом непосредственно не передают семантику основной системе.
>
> *Документация Kubernetes*[1]

Иными словами, метки существуют для маркирования ресурсов с помощью информации, имеющей смысл для нас, но не для Kubernetes. Например, pod-оболочкам часто присваивают метку в виде имени приложения, которому они принадлежат:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: demo
```

Сами по себе метки ничего не делают. Они полезны в качестве информации: кто угодно может взглянуть на pod-оболочку и сразу понять, какое приложение она выполняет. Но настоящую пользу они приносят в сочетании с селекторами.

Селекторы

Селектор — выражение, соответствующее метке (или набору меток). Он предоставляет способ указать группу ресурсов по их меткам. Например, у ресурса Service есть селектор, определяющий pod-оболочки, которым он будет отправлять запросы. Помните наш демонстрационный сервис из подраздела «Ресурсы типа “сервис”» на с. 97?

```
apiVersion: v1
kind: Service
...
spec:
  ...
  selector:
    app: demo
```

Это очень простой селектор, захватывающий все ресурсы, у которых метка `app` равна `demo`. Если у ресурса вообще нет метки `app`, селектор его проигнорирует. То же

[1] kubernetes.io/docs/concepts/overview/working-with-objects/labels.

самое произойдет, если значение метки `app` не равно `demo`. Требованию отвечают только ресурсы (в данном случае pod-оболочки) с меткой `app: demo`: все они будут выбраны этим сервисом.

Метки применяются не только для соединения сервисов и pod-оболочек, вы можете использовать их напрямую в сочетании с флагом `--selector`, когда обращаетесь к кластеру с помощью команды `kubectl get`:

```
kubectl get pods --all-namespaces --selector app=demo
NAMESPACE   NAME                    READY     STATUS    RESTARTS   AGE
demo        demo-5cb7d6bfdd-9dckm   1/1       Running   0          20s
```

Как вы помните, в подразделе «Использование коротких флагов» на с. 158 было показано, что `--selector` можно сократить до `-l` (от англ. *labels*).

Если вы хотите узнать, какие метки назначены вашим pod-оболочкам, передайте команде `kubectl get` флаг `--show-labels`:

```
kubectl get pods --show-labels
NAME                    ... LABELS
demo-5cb7d6bfdd-9dckm   ... app=demo,environment=development
```

Более сложные селекторы

В большинстве случаев вам будет достаточно простых селекторов наподобие `app: demo` (их еще называют *селекторами равенства*). Чтобы сделать селектор более точным, можно объединить несколько разных меток:

```
kubectl get pods -l app=demo,environment=production
```

Команда вернет только те pod-оболочки, у которых есть сразу *две* метки: `app: demo` и `environment: production`. В формате YAML (например, в сервисе) это будет выглядеть так:

```
selector:
  app: demo
  environment: production
```

Сервисы поддерживают только подобные селекторы равенства, но для интерактивных запросов с использованием `kubectl` и более сложных ресурсов вроде развертываний существуют и другие варианты.

Один из них — селектор *неравенства*:

```
kubectl get pods -l app!=demo
```

Эта команда вернет все pod-оболочки, у которых метка `app` не равна `demo` или просто отсутствует.

Вы также можете запрашивать значения меток, входящих в *последовательность*:

```
kubectl get pods -l environment in (staging, production)
```

Эквивалент в формате YAML выглядит так:

```
selector:
  matchExpressions:
  - {key: environment, operator: In, values: [staging, production]}
```

У вас есть и возможность выбирать по меткам, которые *не* входят в заданную последовательность:

```
kubectl get pods -l environment notin (production)
```

Это можно выразить в формате YAML:

```
selector:
  matchExpressions:
  - {key: environment, operator: NotIn, values: [production]}
```

Еще один пример использования `matchExpressions` — в пункте «Использование принадлежности к узлам для управления планированием» на с. 132.

Дополнительные способы использования меток

Мы показали, как связывать pod-оболочки с сервисами с помощью метки `app` (на самом деле можно указать любую метку, просто `app` используется чаще всего). Для чего еще можно применять метки?

В нашем чарте Helm для демонстрационного приложения (см. подраздел «Что внутри у чарта Helm» на с. 281) мы задали метку `environment`, которая, к примеру, может быть равна `staging` или `production`. Если ваши оболочки получают промышленную нагрузку и проходят заключительное тестирование в одном и том же кластере (см. пункт «Нужно ли мне больше одного кластера?» на с. 140), вы можете использовать метки, чтобы выделить две разные среды. Селектор вашего сервиса для выбора промышленных ресурсов мог бы выглядеть так:

```
selector:
  app: demo
  environment: production
```

Без дополнительного селектора `environment` сервис выбрал бы все pod-оболочки с меткой `app: demo`, включая те, что проходят финальное тестирование. Вряд ли вы этого хотите.

В зависимости от того, какие у вас приложения, метки можно использовать для разделения ресурсов разными способами. Вот некоторые примеры:

```
metadata:
  labels:
    app: demo
    tier: frontend
    environment: production
    environment: test
    version: v1.12.0
    role: primary
```

Это позволяет обращаться к кластеру на нескольких разных уровнях, чтобы понять, что происходит.

Метки также можно использовать как средство выкатывания канареечных развертываний (см. подраздел «Канареечные развертывания» на с. 306). Чтобы выкатить новую версию приложения лишь на небольшое количество pod-оболочек, можно использовать такие метки, как `track: stable` и `track: canary`, для двух отдельных развертываний.

Если селектор вашего сервиса захватывает только метку `app`, он пошлет трафик ко всем подходящим pod-оболочкам, включая `stable` и `canary`. Вы можете поменять число реплик в каждом из этих двух развертываний, плавно увеличивая долю pod-оболочек `canary`. После того как все `Pod`-объекты будут переведены на версию `canary`, измените их метки на `stable` и начните процесс заново со следующей версии.

Метки и аннотации

Вам, наверное, интересно, какая разница между метками и аннотациями. И те и другие являются парами типа «ключ — значение» и предоставляют данные о ресурсах.

Отличие в том, что *метки идентифицируют ресурсы*. Они используются для выбора групп связанных между собой ресурсов, как в селекторе сервиса. Аннотации же не предназначены для идентификационной информации и применяются инструментами и сервисами за пределами Kubernetes. В подразделе «Хуки Helm»

на с. 307 показано, как с их помощью можно управлять рабочими процессами в Helm.

Поскольку метки часто участвуют во внутренних запросах, критически важных для производительности Kubernetes, они связаны довольно жесткими ограничениями. Например, имена меток не могут быть длиннее 63 символов, хотя для них можно указать необязательный 253-символьный префикс в виде поддомена DNS, который отделен от метки символом косой черты. Метки должны начинаться только с алфавитно-цифровых символов (букв или цифр), а кроме них, могут содержать тире, подчеркивания и точки. Значения меток ограничиваются похожим образом (kubernetes.io/docs/concepts/overview/working-with-objects/labels/#syntax-and-character-set).

Но на практике вам вряд ли когда-нибудь не хватит символов для меток, так как большинство из них — это одно слово (например, `app`).

Принадлежность к узлам

Мы упоминали принадлежность к узлам в пункте «Использование принадлежности к узлам для управления планированием» на с. 132, когда обсуждали прерываемые серверы. Там вы научились использовать эту возможность для высоко- и низкоприоритетного размещения pod-оболочек на определенных узлах. Далее поговорим о ней более подробно.

В большинстве случаев принадлежность к узлам вам будет не нужна: Kubernetes умеет довольно разумно распределять pod-оболочки между подходящими узлами. Если все ваши узлы одинаково подходят для выполнения той или иной pod-оболочки, об этом можете не беспокоиться.

Но так бывает не всегда (как в предыдущем примере с прерываемыми серверами). Если перезапуск `Pod`-объекта отнимает много ресурсов, лучше не размещать его на прерываемых узлах, потому что те имеют особенность исчезать из кластера без предупреждения. Подобное предпочтение можно выразить в виде принадлежности к узлам.

Принадлежность может быть жесткой и мягкой. Инженеры, как правило, не блещут талантами в выборе изящных названий, поэтому в Kubernetes используются следующие термины:

- `requiredDuringSchedulingIgnoredDuringExecution` (жесткая);
- `preferredDuringSchedulingIgnoredDuringExecution` (мягкая).

Возможно, вам лучше запомнить, что `required` (обязательная) относится к жесткой принадлежности (правило *должно* соблюдаться при планировании pod-оболочки), а `preferred` (предпочтительная) — к мягкой (было бы *здорово*, если бы правило соблюдалось, но это не столь важно).

Длинные имена жесткой и мягкой принадлежностей указывают на то, что эти правила применяются *во время планирования*, но не *во время выполнения*. То есть уже запланированная для выполнения на определенном узле с подходящей принадлежностью pod-оболочка там и останется. Если во время работы что-то изменится и правило перестанет соблюдаться, Kubernetes не будет перемещать pod-оболочку (эта возможность может появиться в будущем).

Жесткая принадлежность

Принадлежность описывает типы узлов, на которых вы хотите запустить свою pod-оболочку. Вы можете указать несколько правил для платформы Kubernetes, по которым она должна выбирать узлы. Каждое из них выражается в виде поля `nodeSelectorTerms`. Например:

```
apiVersion: v1
kind: Pod
...
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: "failure-domain.beta.kubernetes.io/zone"
            operator: In
            values: ["us-central1-a"]
```

Данному правилу соответствуют только узлы в зоне `us-central1-a`, следовательно, работа pod-оболочки будет планироваться только в этой зоне.

Мягкая принадлежность

Мягкая принадлежность выражается аналогичным образом, за исключением того, что каждому правилу назначается *вес* в виде числа от 1 до 100, который определяет влияние правила на результат. Например:

```
preferredDuringSchedulingIgnoredDuringExecution:
- weight: 10
```

```
  preference:
    matchExpressions:
    - key: "failure-domain.beta.kubernetes.io/zone"
      operator: In
      values: ["us-central1-a"]
- weight: 100
  preference:
    matchExpressions:
    - key: "failure-domain.beta.kubernetes.io/zone"
      operator: In
      values: ["us-central1-b"]
```

Поскольку это правило начинается с `preferred...`, оно означает мягкую принадлежность: Kubernetes может разместить pod-оболочку на любом узле, но приоритет будет отдавать тем из них, которые соответствуют данным правилам.

Как видите, эти два правила имеют разные значения `weight`. У первого вес 10, а у второго — 100. Если есть узлы, соответствующие обоим правилам, Kubernetes отдаст десятикратный приоритет узлам, находящимся в зоне доступности `us-central1-b` (второе правило).

Вес помогает выразить относительную важность ваших предпочтений.

Принадлежность и непринадлежность pod-оболочек

Мы уже видели, как с помощью принадлежности можно склонить планировщик в ту или иную сторону при выборе для pod-оболочки узлов определенных типов. Но возможно ли повлиять на планирование, исходя из того, какие pod-оболочки *уже* находятся на узле?

Некоторые `Pod`-объекты работают лучше, будучи размещенными на одном узле: например, веб-сервер и система кэширования содержимого, такая как Redis. Было бы полезно иметь возможность добавлять в спецификацию pod-оболочки информацию, которая бы описывала ее предпочтения относительно других `Pod`-объектов с определенным набором меток.

С другой стороны, некоторым pod-оболочкам лучше не встречаться. В подразделе «Балансировка вашей рабочей нагрузки» на с. 133 мы видели, какого рода проблемы могут возникнуть, если реплики pod-оболочки очутятся на одном узле вместо того, чтобы быть распределенными по кластеру. Можно ли попросить планировщик избегать размещения `Pod`-объекта там, где уже находится его реплика?

Именно для этого принадлежность pod-оболочек и предназначена. Как и принадлежность к узлам, она описывается в виде правил: либо жестких требований, либо предпочтений с весами.

Размещение pod-оболочек вместе

Сначала рассмотрим первый случай — размещения pod-оболочек вместе. Представьте, что у вас есть две pod-оболочки с метками `app: server` и `app: cache`: первая содержит веб-сервер, а вторая — систему кэширования. Они могут взаимодействовать, находясь на разных узлах, но лучше их объединить, чтобы им не нужно было общаться по сети. Как попросить об этом у планировщика?

Вот пример жесткой принадлежности pod-оболочки, выраженной в виде спецификации `server`. Эффект будет таким же, как если бы вы добавили эти параметры в спецификацию pod-оболочки `cache` (либо в спецификации обеих сразу):

```
apiVersion: v1
kind: Pod
metadata:
  name: server
  labels:
    app: server
...
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
        - matchExpressions:
          - key: app
            operator: In
            values: ["cache"]
        topologyKey: kubernetes.io/hostname
```

В целом эта принадлежность делает так, что работа pod-оболочки `server` по мере возможностей планируется на узле, где уже выполняется pod-оболочка с меткой `cache`. Если такого узла нет или у подходящих узлов недостаточно свободных ресурсов, pod-оболочка не сможет запуститься.

Вряд ли вам понадобится именно такое поведение в реальных условиях: чтобы две pod-оболочки совершенно точно находились вместе, поместите их контейнеры в один `Pod`-объект. Если такое размещение является всего лишь предпочтительным, используйте мягкую принадлежность (`preferredDuringSchedulingIgnoredDuringExecution`).

Размещение pod-оболочек порознь

Теперь рассмотрим пример непринадлежности: когда определенные pod-оболочки должны размещаться отдельно. Вместо `podAffinity` используем `podAntiAffinity`:

```
apiVersion: v1
kind: Pod
metadata:
  name: server
  labels:
    app: server
...
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
        - matchExpressions:
          - key: app
            operator: In
            values: ["server"]
        topologyKey: kubernetes.io/hostname
```

Этот пример очень похож на предыдущий, только здесь указана *непринадлежность* и она выражает противоположный смысл. Таким образом, сопоставляемое выражение отличается и в данном случае выглядит так: «Метка `app` должна иметь значение `server`».

В результате работа pod-оболочки *не* будет планироваться на узлах, которые соответствуют этому правилу. Иными словами, никакая pod-оболочка с меткой `app: server` не может быть размещена на узле, где находится другая pod-оболочка с такой же меткой. Это обеспечивает равномерное распределение pod-оболочек `server` по кластеру, возможно, в ущерб желаемому количеству реплик.

Мягкая непринадлежность

Однако обычно нас больше заботит наличие достаточного числа реплик, чем их максимально равномерное распределение. Жесткое правило не совсем то, что нам надо в данной ситуации. Немного изменим его, превратив в мягкую непринадлежность:

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
    - weight: 1
      podAffinityTerm:
        labelSelector:
```

```
      - matchExpressions:
        - key: app
          operator: In
          values: ["server"]
      topologyKey: kubernetes.io/hostname
```

Обратите внимание, что теперь правило начинается с `preferred...`, а не с `required...`, и таким образом описывает мягкую непринадлежность. Если это возможно, правило будет выполнено, если нет — Kubernetes все равно запланирует работу pod-оболочки.

Поскольку речь идет о предпочтении, мы указываем значение `weight`, как делали и с мягкой принадлежностью к узлам. Если указать несколько правил принадлежности, их приоритет будет соответствовать их весу.

Когда использовать правила принадлежности pod-оболочек

Правила принадлежности, относящиеся как к pod-оболочкам, так и к узлам, следует рассматривать в качестве тонкой настройки в особых случаях. Планировщик и сам по себе хорошо справляется с размещением pod-оболочек, обеспечивая наилучшие производительность и доступность кластера. Принадлежность pod-оболочек ограничивает свободу планировщика, заставляя его отдавать предпочтение одним приложениям перед другими. Эту возможность следует использовать в ситуации, когда только так можно решить проблему, возникшую в промышленных условиях.

Ограничения и допуски

Из раздела «Принадлежность к узлам» на с. 210 вы узнали о свойстве pod-оболочек, которое может склонить их к той или иной группе узлов (или отвернуть от нее). В свою очередь, *ограничения* (taints) позволяют узлу отвергнуть группу pod-оболочек, руководствуясь свойствами в своей спецификации.

Ограничения, к примеру, можно использовать для создания выделенных узлов, зарезервированных только для конкретного вида pod-оболочек. Kubernetes будет создавать ограничения автоматически при возникновении определенных проблем на узле, таких как нехватка памяти или отсутствие сетевого соединения.

Чтобы добавить ограничение для отдельного узла, используйте команду `kubectl taint`:

```
kubectl taint nodes docker-for-desktop dedicated=true:NoSchedule
```

Это создаст ограничение под названием `dedicated=true` на узле `docker-for-desktop`. Результат будет тот же, что и у `NoSchedule`: там можно будет размещать только pod-оболочки с таким же *ограничением*.

Чтобы вывести ограничения, установленные для определенного узла, используйте команду `kubectl describe node...`.

Чтобы убрать ограничение на узле, повторите команду `kubectl taint`, но укажите в конце после его имени знак «минус»:

```
kubectl taint nodes docker-for-desktop dedicated:NoSchedule-
```

Допуски (tolerations) — это свойства pod-оболочки, описывающие ограничения, с которыми те совместимы. Например, чтобы pod-оболочка допускала ограничение `dedicated=true`, добавьте следующий код в ее спецификацию:

```
apiVersion: v1
kind: Pod
...
spec:
  tolerations:
  - key: "dedicated"
    operator: "Equal"
    value: "true"
    effect: "NoSchedule"
```

Этим мы, по сути, говорим: «Данной pod-оболочке разрешается работать на узлах с ограничением `dedicated=true`, которое имеет тот же эффект, что и `NoSchedule`». Поскольку допуск *совпадает* с ограничением, работа pod-оболочки может быть запланирована. Ни одному `Pod`-объекту без этого допуска не будет позволено выполняться на ограниченном узле.

Если в результате ограничения pod-оболочка вообще не может быть развернута, она будет оставаться в состоянии `Pending` и в ее описании вы сможете увидеть следующее сообщение:

```
Warning FailedScheduling 4s (x10 over 2m) default-scheduler 0/1 nodes are
available: 1 node(s) had taints that the pod didn't tolerate.
```

Ограничения и допуски имеют и другие применения, среди которых маркировка узлов со специализированным оборудованием (таким как графические адаптеры) и разрешение каким-то pod-оболочкам допускать определенного рода проблемы с узлами.

Например, если узел отключится от сети, Kubernetes автоматически добавит для него ограничение `node.kubernetes.io/unreachable`. Обычно это приводит к тому, что `kubelet` выселяет все pod-оболочки узла. Но, возможно, вы надеетесь на то, что

доступ к сети вернется в разумные сроки, и хотите, чтобы какие-то pod-оболочки продолжали работать. Для этого вы можете добавить в спецификацию соответствующих `Pod`-объектов допуск, совпадающий с ограничением `unreachable`.

Больше об ограничениях и допусках можно прочитать в документации Kubernetes (kubernetes.io/docs/concepts/configuration/taint-and-toleration).

Контроллеры pod-оболочек

В этой главе pod-оболочкам уделяется много внимания, и неспроста: внутри них выполняются все приложения в Kubernetes. Но вам, наверное, интересно, зачем тогда нужны другие виды объектов? Разве не достаточно просто создать `Pod`-объект для приложения и запустить его?

Это вы делаете, запуская контейнер напрямую с помощью команды `docker container run`, как делали в подразделе «Запуск образа контейнера» на с. 53. Подход вполне рабочий, но очень ограниченный:

- если контейнер по какой-либо причине завершит работу, вы должны перезапустить его вручную;
- ваш контейнер имеет лишь одну реплику, а если создать вручную несколько реплик, вы не сможете распределять между ними трафик;
- если вам нужны высокодоступные реплики, вы должны решать, на каких узлах их размещать, и заботиться о балансировке кластера;
- при обновлении контейнера вам придется по очереди останавливать, заменять и перезапускать каждый активный образ.

Это та работа, которую Kubernetes выполняет за вас с помощью *контроллеров*. В разделе «Объекты ReplicaSet» на с. 93 мы познакомили вас с контроллером `ReplicaSet`, управляющим группой реплик определенной pod-оболочки. Он непрерывно следит за тем, чтобы в кластере всегда было заданное количество реплик: если их недостаточно, запускает новые, а если слишком много, удаляет существующие.

Вы также знакомы с развертываниями, которые, как описывалось в одноименном разделе на с. 89, управляют объектами `ReplicaSet`, контролируя выкатывание обновлений приложения. Когда вы обновляете развертывание (скажем, изменяя спецификацию контейнера), оно создает новый объект `ReplicaSet` для запуска новых pod-оболочек и в итоге удаляет его старый экземпляр, управлявший старыми `Pod`-объектами.

Для большинства простых приложений развертываний вполне достаточно. Однако существует несколько других полезных разновидностей pod-контроллеров, и некоторые из них мы кратко рассмотрим далее.

Объекты DaemonSet

Представьте, что вам нужно отправлять журнальные записи всех ваших приложений на центральный журнальный сервер, как это делают системы мониторинга в продуктах SaaS, такие как Datadog (см. подраздел «Datadog» на с. 374) или стек Elasticsearch-Logstash-Kibana (ELK).

Вы можете добавить в каждое приложение код, который будет подключаться к сервису ведения журнала, аутентифицироваться, сохранять журнальные записи и т. д. Но это приведет к дублированию большой части кода, что довольно неэффективно.

Вы также могли бы добавить в каждую pod-оболочку по одному дополнительному контейнеру, который играл бы роль агента для ведения журнала (шаблон проектирования «Прицеп»). Это означает, что приложениям не нужно будет знать о том, как общаться с сервером ведения журнала, а вы сможете получить несколько копий агента на каждом узле.

На самом деле вам нужна только одна копия, так как данный агент отвечает лишь за соединение с сервисом ведения журнала и передачу ему журнальных записей. Это настолько распространенное действие, что в Kubernetes для него предусмотрен специальный контроллер — `DaemonSet`.

Термин daemon (демон) традиционно относится к длительным фоновым процессам на сервере, которые занимаются такими делами, как ведение журнала. Поэтому контроллер DaemonSet в Kubernetes по аналогии запускает на каждом узле кластера контейнер-демон.

Как и следовало ожидать, манифест `DaemonSet` очень похож на манифест развертывания:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  ...
spec:
  ...
```

```
template:
  ...
  spec:
    containers:
    - name: fluentd-elasticsearch
      ...
```

Используйте `DaemonSet`, когда вам нужно запустить по одной копии pod-оболочки на каждом узле кластера. Если для вашего приложения поддержание заданного количества реплик важнее, чем то, на каких именно узлах они размещаются, используйте развертывания.

Объект StatefulSet

Подобно развертыванию и `DaemonSet`, объект `StatefulSet` является разновидностью контроллера pod-оболочек. Его отличительная черта — возможность запускать и останавливать pod-оболочки в определенной последовательности.

Например, при использовании развертываний порядок запуска и остановки ваших `Pod`-объектов произвольный. Это подходит для сервисов, не сохраняющих состояние, когда все реплики являются идентичными и выполняют одну и ту же работу.

Но иногда pod-оболочки необходимо запускать в определенной последовательности с возможностью идентифицировать их по порядковому номеру. Например, распределенные приложения, такие как Redis, MongoDB или Cassandra, которые создают свои собственные кластеры и имеют потребность как-то определять лидирующий узел по его предсказуемому имени.

Для этого идеально подходит `StatefulSet`. Если вы, к примеру, создадите объект `StatefulSet` с именем `redis`, первая запущенная pod-оболочка будет называться `redis-0`: Kubernetes подождет, пока она будет готова к работе, и только потом запустит следующую, `redis-1`.

В зависимости от приложения эту возможность можно использовать для предсказуемой кластеризации pod-оболочек. Например, каждый `Pod`-объект может запускать стартовый скрипт, который проверяет, выполняется ли он на `redis-0`. Если ответ положительный, объект становится лидером кластера. Если нет, он попытается присоединиться к кластеру, связавшись с `redis-0`.

Чтобы запустить следующую реплику в `StatefulSet`, Kubernetes необходимо убедиться в том, что предыдущая уже готова к работе. При уничтожении `StatefulSet` реплики останавливаются в обратном порядке, завершая свою работу по очереди.

В целом, если не принимать во внимание эти особенности, контроллер `StatefulSet` очень похож на обычное развертывание:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  selector:
    matchLabels:
      app: redis
  serviceName: "redis"
  replicas: 3
  template:
    ...
```

Чтобы вы могли обращаться к pod-оболочкам по предсказуемым доменным именам, таким как `redis-1`, вам, кроме всего прочего, нужно создать сервис с типом `clusterIP`, равным `None` (так называемый *неуправляемый сервис*).

В случае с управляемым сервисом вы получаете единую DNS-запись (такую как `redis`), которая распределяет нагрузку между всеми внутренними pod-оболочками. Если сервис неуправляемый, вы получаете такое же единое доменное имя, но вместе с этим каждой пронумерованной pod-оболочке выдается отдельная DNS-запись вида `redis-0`, `redis-1`, `redis-2` и т. д.

Pod-оболочки, которым необходимо присоединиться к кластеру Redis, могут обращаться прямо к `redis-0`. Но если приложению нужен просто сервис Redis с балансировкой нагрузки, оно может использовать доменное имя `redis`, чтобы общаться с произвольно выбираемыми pod-оболочками.

Контроллеры `StatefulSet` также могут управлять дисковыми хранилищами своих pod-оболочек, используя объект `VolumeClaimTemplate`, который автоматически создает экземпляры `PersistentVolumeClaim` (см. подраздел «Постоянные тома» на с. 201).

Запланированные задания

Еще одной полезной разновидностью pod-контроллера в Kubernetes является запланированное задание (`Job`-объект). В отличие от развертываний, которые запускают определенное количество pod-оболочек и непрерывно их перезапускают, запланированные задания запускают `Pod`-объекты конкретно столько раз, сколько вам нужно. После этого задание считается выполненным.

Представьте себе пакетную обработку или рабочую pod-оболочку в очереди: она стартует, делает свою работу и затем останавливается. Это идеально подходит для оформления в виде запланированного задания.

Выполнение задания управляется двумя полями: `completions` и `parallelism`. Первое, `completions`, определяет, сколько раз заданная pod-оболочка должна успешно отработать, прежде чем задание можно будет считать выполненным. По умолчанию значение равно `1`, что означает однократное выполнение.

Поле `parallelism` указывает, сколько pod-оболочек должно работать одновременно. Значение по умолчанию равно `1`, поэтому pod-оболочка будет запускаться в единственном экземпляре.

Представьте, к примеру, что вам нужно выполнить рабочее задание, которое будет доставать рабочие элементы из очереди. Вы можете присвоить полю `parallelism` значение `10`, оставив `completions` как есть (то есть `1`). Таким образом, запустится десять pod-оболочек, и каждая из них начнет обращаться в очередь за элементами: когда очередь опустеет, pod-оболочки остановятся, а запланированное задание будет считаться завершенным:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: queue-worker
spec:
  completions: 1
  parallelism: 10
  template:
    metadata:
      name: queue-worker
    spec:
      containers:
        ...
```

Еще можно представить ситуацию, когда вам понадобится выполнить что-то наподобие пакетной обработки. В этом случае можете оставить значение `1` для обоих полей, `completions` и `parallelism`. Kubernetes запустит одну копию pod-оболочки и дождется ее успешного завершения. Если она откажет или завершит работу неудачно, запланированное задание перезапустит ее, как и при развертывании. В поле `completions` учитываются только успешные случаи завершения работы.

Как же запустить запланированное задание? Это можно сделать вручную, применив его манифест с помощью `kubectl` или Helm. Но задание может срабатывать и автоматически, например в рамках непрерывного развертывания (см. главу 14).

Наверное, самым популярным способом является периодический запуск запланированных заданий: в определенное время суток или с заданным интервалом. У Kubernetes для этого предусмотрен специальный тип запланированных заданий — Cronjob.

Задания Cronjob

В средах Unix планируемые задания выполняются демоном `cron` (чье название происходит от греческого слова χρόνος — «время»). Соответственно их называют *cron jobs*. В Kubernetes объект Cronjob занимается тем же самым.

Задание Cronjob выглядит так:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: demo-cron
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      ...
```

В манифесте Cronjob есть два важных поля, на которые стоит обратить внимание: `spec.schedule` и `spec.jobTemplate`. Поле `schedule` определяет, когда задание нужно запускать: в нем используется тот же формат, что и в Unix-утилите `cron`.

Поле `jobTemplate` описывает шаблон для задания, которое нужно запустить. Это поле ничем не отличается от аналогичного в манифесте `Job`-объекта (см. подраздел «Запланированные задания» на с. 220).

Горизонтальное автомасштабирование pod-оболочек

Как вы помните, контроллер развертывания поддерживает в рабочем состоянии определенное количество копий pod-оболочек. Если одна реплика откажет, другая будет запущена вместо нее. А если по какой-то причине было создано слишком много `Pod`-объектов, развертывание остановит лишние, чтобы достичь желаемого числа реплик.

Это желаемое число прописывается в манифесте развертывания. Как мы уже видели, его можно увеличивать, чтобы получить больше pod-оболочек при высокой нагрузке, или уменьшать, чтобы сократить развертывание в случае, когда некоторые pod-оболочки простаивают.

Но что, если бы платформа Kubernetes сама могла автоматически регулировать количество реплик в зависимости от нагрузки? Именно этим и занимается контроллер горизонтального автомасштабирования pod-оболочек (*горизонтальное* масштабирование изменяет число реплик сервиса, тогда как *вертикальное* делает отдельные реплики больше или меньше).

Horizontal Pod Autoscaler (HPA) наблюдает за указанным развертыванием, отслеживая определенные показатели, и соответственно увеличивает или уменьшает количество реплик.

Одним из самых распространенных показателей автомасштабирования является загруженность процессора. Как вы помните из подраздела «Запросы ресурсов» на с. 108, pod-оболочки могут запрашивать определенное количество вычислительных ресурсов, например 500 миллипроцессоров. В ходе работы pod-оболочки загруженность процессора колеблется. Это означает, что в любой момент времени pod-оболочка на самом деле использует только некоторую часть от своего исходного запроса.

Вы можете автомасштабировать развертывания, опираясь на это значение: например, создать контроллер HPA, который нацелен на 80%-ную загруженность процессора. Например, когда все pod-оболочки в среднем будут занимать 70 % от запрошенного объема, HPA начнет сокращать количество реплик: ведь если pod-оболочки не работают на полную, их столько не нужно.

А когда средняя загруженность превысит выбранное вами значение и достигнет 90 %, вам нужно будет ее снизить, добавив больше реплик: HPA модифицирует развертывание и увеличит количество pod-оболочек.

Каждый раз, когда контроллер HPA видит необходимость в масштабировании, он регулирует число реплик, исходя из соотношения показателей текущей и желаемой загруженности. Если развертывание находится очень близко к желаемой загруженности, HPA добавит или удалит лишь небольшое количество реплик, если же разница значительная, масштаб тоже будет откорректирован существенным образом.

Вот пример HPA на основе загруженности процессора:

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: demo-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: demo
  minReplicas: 1
  maxReplicas: 10
```

```
metrics:
- type: Resource
  resource:
    name: cpu
    targetAverageUtilization: 80
```

Обратите внимание на следующие поля:

- `spec.scaleTargetRef` указывает развертывание, которое нужно масштабировать;
- `spec.minReplicas` и `spec.maxReplicas` задают границы масштабирования;
- `spec.metrics` определяет показатели, которые будут использоваться для масштабирования.

Показатель загруженности процессора используется чаще всего, но вы можете указать любые другие, доступные в Kubernetes, — как *системные* (показатели процессора и памяти), встроенные в платформу, так и *служебные*, которые вы определяете и экспортируете из своего приложения (см. главу 16). Например, вы можете менять масштаб в зависимости от частоты возникновения ошибок в вашем коде.

Больше об автомасштабировании и пользовательских показателях читайте в документации Kubernetes (kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough).

PodPreset

`PodPreset` позволяет внедрять информацию в pod-оболочки в момент их создания. Это экспериментальная возможность Kubernetes, находящаяся на стадии альфа-тестирования. Вы можете, к примеру, создать объект `PodPreset`, который подключает том ко всем pod-оболочкам с определенным набором меток.

Объект `PodPreset` представляет собой так называемый *контроллер доступа*. Такие контроллеры отслеживают создание pod-оболочек и предпринимают какие-то действия, если те соответствуют их селекторам. Например, некоторые контроллеры доступа могут блокировать создание pod-оболочек, нарушающих какую-нибудь политику, тогда как другие, вроде `PodPreset`, внедряют в pod-оболочки дополнительную конфигурацию.

Ниже показан пример объекта `PodPreset`, который добавляет том `cache` ко всем pod-оболочкам, соответствующим селектору `tier: frontend`:

```
apiVersion: settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: add-cache
```

```
spec:
  selector:
    matchLabels:
      role: frontend
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

Параметры, определяемые объектом `PodPreset`, объединяются с параметрами pod-оболочки. Экземпляры `Pod`, модифицированные таким образом, имеют аннотацию следующего вида:

```
podpreset.admission.kubernetes.io/podpreset-add-cache: "<resource version>"
```

Что происходит, когда собственные параметры pod-оболочки конфликтуют с теми, что описаны в `PodPreset`, или когда конфликтующая конфигурация указана в нескольких объектах `PodPreset`? Тогда Kubernetes отказывается модифицировать pod-оболочку, а в ее описании появляется сообщение `Conflict on pod preset`.

В связи с этим `PodPreset` нельзя использовать для переопределения существующей конфигурации pod-оболочки: заполнять можно только не заданные в ней настройки. Если есть необходимость защитить pod-оболочку от модификации со стороны `PodPreset`, укажите аннотацию:

```
podpreset.admission.kubernetes.io/exclude: "true"
```

Поскольку объекты `PodPreset` все еще являются экспериментальными, они могут оказаться недоступными в управляемых кластерах Kubernetes. Но и в самостоятельно размещаемых кластерах для их включения, вероятно, потребуются дополнительные действия, такие как передача аргументов командной строки серверу API. Подробнее об этом — в документации Kubernetes (kubernetes.io/docs/concepts/workloads/pods/podpreset).

Операторы и определение пользовательских ресурсов

В подразделе «Объект StatefulSet» на с. 219 мы уже видели, что, хотя стандартные объекты Kubernetes, такие как развертывания и сервисы, хорошо подходят для простых приложений, не сохраняющих состояние, они имеют свои ограничения. Некоторым приложениям нужно несколько взаимодействующих между собой pod-оболочек, инициализированных в определенном порядке (например, реплицируемые базы данных или кластеризированные сервисы).

Если приложению требуется более сложный контроль, чем тот, что предоставляет `StatefulSet`, вы можете создать свои собственные новые типы объектов, так называемые *определения пользовательских ресурсов* (Custom Resource Definition, CRD). Например, инструмент для резервного копирования Velero создает пользовательские объекты Kubernetes, такие как `Config` и `Backup` (см. подраздел «Velero» на с. 268).

Платформа Kubernetes изначально является расширяемой, поэтому вы можете свободно определять и создавать любые типы объектов по вашему усмотрению, используя механизм CRD. Некоторые объекты CRD, такие как Velero `BackupStorageLocation`, предназначены лишь для хранения данных. Но если пойти дальше, реально создать и объекты, действующие как pod-контроллеры, развертывания или `StatefulSet`.

Например, если бы вам было нужно создать контроллер, который подготавливает в Kubernetes реплицируемые высокодоступные кластеры с базой данных MySQL, как бы вы к этому подошли?

Прежде всего потребовалось бы создать определение CRD для своего пользовательского контроллера. Чтобы он мог выполнять какие-то действия, необходимо написать программу, взаимодействующую с Kubernetes API. Как мы уже видели в разделе «Создание собственных инструментов для работы с Kubernetes» на с. 179, это не так уж и сложно. Подобная программа называется *оператором* (возможно, потому, что она автоматизирует действия, которые мог бы выполнять реальный человек-оператор).

Для написания оператора не нужны никакие пользовательские объекты. Инженер DevOps Майкл Тричер продемонстрировал хороший пример оператора, который следит за созданием новых пространств имен и добавляет к каждому из них `RoleBinding` (см. подраздел «Введение в управление доступом на основе ролей» на с. 258).

Но обычно операторы используют один или несколько пользовательских объектов, созданных посредством CRD. Поведение этих объектов затем реализуется программой, взаимодействующей с Kubernetes API.

Ресурсы Ingress

Ingress — это своего рода балансировщик нагрузки, размещенный перед сервисом (рис. 9.1). Он передает сервису запросы, поступающие от клиентов. Сервис, в свою очередь, отправляет их подходящим pod-оболочкам, основываясь на селекторе меток (см. подраздел «Ресурсы типа “сервис”» на с. 97).

Вот пример очень простого ресурса Ingress:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: demo-ingress
spec:
  backend:
    serviceName: demo-service
    servicePort: 80
```

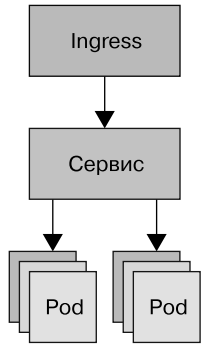

Рис. 9.1. Ресурс Ingress

Ingress направляет трафик к сервису с именем `demo-service` на порте 80 (на самом деле запросы идут напрямую от Ingress к подходящим pod-оболочкам, но концептуально лучше считать, что они проходят через сервис).

Сам по себе этот пример не кажется слишком полезным. Но Ingress может делать куда больше.

Правила Ingress

Сервисы хорошо подходят для маршрутизации *внутреннего* трафика вашего кластера (например, от одного микросервиса к другому). Объекты Ingress, в свою очередь, предназначены для перенаправления *внешних запросов* к вашему кластеру и подходящему микросервису.

Ingress может направлять трафик к разным сервисам, исходя из установленных вами правил. Наиболее распространена маршрутизация запросов в зависимости от их URL-адресов (это называют *разветвлением*):

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: fanout-ingress
spec:
  rules:
  - http:
    paths:
    - path: /hello
      backend:
        serviceName: hello
        servicePort: 80
    - path: /goodbye
      backend:
        serviceName: goodbye
        servicePort: 80
```

У разветвления есть множество применений. Высокодоступные балансировщики бывают достаточно дорогими, а с помощью разветвления Ingress и одного балансировщика нагрузки можно распределять трафик по большому количеству сервисов.

Вы не ограничены только маршрутизацией по URL и можете использовать HTTP-заголовок `Host` (эквивалент методики, известной как *виртуальный хостинг с доступом по имени*). Запросы к сайтам с разными доменами (такими как `example.com`) направляются к соответствующим внутренним сервисам.

Терминация TLS с помощью Ingress

Помимо прочего, Ingress поддерживает защищенные соединения на основе протокола TLS (ранее известного как SSL). Если у вас много разных сервисов и приложений на одном домене, у них у всех может быть общий TLS-сертификат и управлять этими соединениями может один ресурс Ingress (это называют *терминацией TLS*):

```
apiVersion: extensions/v1beta1

kind: Ingress
metadata:
  name: demo-ingress
spec:
  tls:
  - secretName: demo-tls-secret
  backend:
    serviceName: demo-service
    servicePort: 80
```

Здесь мы добавили новый раздел `tls`, который заставляет Ingress использовать TLS-сертификат для защиты взаимодействия с клиентами. Сам сертификат хранится в виде ресурса Kubernetes `Secret` (см. раздел «Конфиденциальные данные в Kubernetes» на с. 243).

Использование существующих TLS-сертификатов

Если у вас уже есть TLS-сертификат или вы собираетесь его приобрести в центре сертификации, имейте в виду, что его можно будет использовать вместе с вашим ресурсом Ingress. Создайте объект `Secret` следующего вида:

```
apiVersion: v1
kind: Secret
type: kubernetes.io/tls
```

```
metadata:
  name: demo-tls-secret
data:
  tls.crt: LS0tLS1CRUdJTiBDRV...LS0tCg==
  tls.key: LS0tLS1CRUdJTiBSU0...LS0tCg==
```

Поместите содержимое сертификата в поле `tls.crt`, а ключ — в поле `tls.key`. Как принято в Kubernetes `Secret`, перед добавлением этих данных в манифест их следует закодировать в формате base64 (см. пункт «base64» на с. 246).

Автоматизация сертификатов LetsEncrypt с помощью Cert-Manager

Если вы хотите автоматически запрашивать и обновлять TLS-сертификаты, используя популярный центр сертификации LetsEncrypt (или другой провайдер на основе ACME), обратитесь к инструменту `cert-manager` (docs.cert-manager.io/en/latest).

Если запустить утилиту `cert-manager` внутри кластера, она автоматически найдет ресурсы Ingress с TLS, которые не имеют сертификата, и запросит его у указанного поставщика (например, у LetsEncrypt). Данная утилита является более современным и функциональным наследником популярного инструмента `kube-lego`.

То, как именно ведут себя TLS-соединения, зависит от так называемого *контроллера Ingress*.

Контроллеры Ingress

Контроллер Ingress отвечает за управление одноименными ресурсами в кластере. Выбор контроллера зависит от того, в какой среде размещен ваш кластер.

Обычно для настройки поведения ресурса Ingress к нему добавляются определенные аннотации, которые распознаются контроллером Ingress.

Кластеры, запущенные в Google GKE, могут использовать для Ingress балансировщик Compute Load Balancer. У AWS есть похожий продукт под названием Application Load Balancer. Эти управляемые сервисы предоставляют публичный IP-адрес, на котором Ingress будет отслеживать запросы.

Если вы запускаете Kubernetes на Google Cloud или AWS и хотите использовать Ingress, можете начать с чтения документации к соответствующему продукту, которая находится в следующих репозиториях:

- документация для Google Ingress (github.com/kubernetes/ingress-gce);
- документация для AWS Ingress (github.com/kubernetes-sigs/aws-alb-ingress-controller).

Вы также можете установить и запустить собственный контроллер Ingress внутри своего кластера, а при желании можете использовать сразу несколько контроллеров. Вот некоторые распространенные решения.

- *nginx-ingress* (github.com/nginxinc/kubernetes-ingress). NGINX уже давно стал популярным средством балансировки нагрузки, еще до появления Kubernetes. Контроллер nginx-ingress предоставляет Kubernetes большинство возможностей NGINX. Существуют и другие контроллеры Ingress на основе NGINX, но этот является официальным.
- *Contour* (github.com/projectcontour/contour). Это еще один полезный инструмент для Kubernetes, поддерживаемый компанией Heptio, которую мы еще не раз упомянем в этой книге. На самом деле Contour использует внутри себя другой инструмент под названием Envoy, чтобы проксировать запросы между клиентами и pod-оболочками.
- *Traefik* (docs.traefik.io/user-guide/kubernetes). Это легковесный прокси, который может автоматически управлять TLS-сертификатами для ваших ресурсов Ingress.

Все эти контроллеры имеют разные возможности, конфигурацию, инструкции по установке и подходы к управлению маршрутизацией и сертификатами. Почитайте обо всех вариантах и опробуйте их в своем собственном кластере на своих приложениях, чтобы почувствовать, как они работают.

Istio

Istio — пример технологии, часто называемой *межсервисным взаимодействием*. Она крайне полезна в ситуациях, когда разные команды разрабатывают несколько приложений и сервисов, которые должны сообщаться друг с другом. Она берет на себя маршрутизацию и шифрование сетевого трафика между сервисами, добавляя такие важные функции, как сбор показателей, ведение журнала и балансировка нагрузки.

Istio представляет собой отдельный компонент, который можно подключить ко множеству кластеров Kubernetes, в том числе и к Google Kubernetes Engine (сверьтесь с документацией своего провайдера, чтобы узнать, как включить Istio).

Если вы хотите установить Istio на кластер с самостоятельным размещением, используйте официальный чарт для Helm (istio.io/docs/setup/install/helm).

С Istio особенно рекомендуется ознакомиться в том случае, если ваши приложения в значительной степени зависят друг от друга. Этот инструмент заслуживает отдельной книги, и, скорее всего, он ее получит. Но пока начните с вводной документации (istio.io/docs/concepts/what-is-istio).

Envoy

Большинство управляемых сервисов Kubernetes, таких как Google Kubernetes Engine, предоставляют определенную интеграцию с облачным балансировщиком нагрузки. Например, если создать в GKE сервис типа `LoadBalancer` или ресурс Ingress, платформа автоматически создаст экземпляр Google Cloud Load Balancer и подключит его к вашему сервису.

Хотя эти стандартные облачные балансировщики хорошо масштабируются, они очень просты и не требуют особой настройки. Например, по умолчанию обычно используется алгоритм балансировки `random` (см. подраздел «Ресурсы типа “сервис”» на с. 97). В результате каждое соединение направляется к произвольно выбранному серверу.

Это не всегда то, что нужно. Скажем, если запросы к вашему сервису будут длительными и ресурсоемкими, некоторые из ваших узлов могут оказаться перегруженными, а другие будут простаивать.

Более умный алгоритм мог бы направлять запросы к наименее занятому серверу. Такой алгоритм иногда называют `leastconn` или `LEAST_REQUEST`.

Для подобной более сложной балансировки можно воспользоваться продуктом под названием Envoy (www.envoyproxy.io). Он не входит в состав проекта Kubernetes, но часто используется вместе с его приложениями.

Envoy — это высокопроизводительный распределенный прокси, написанный на C++. Он предназначен для работы с отдельными сервисами и приложениями, но также его можно применять как часть архитектуры межсервисного взаимодействия (см. раздел «Istio» на с. 230).

Разработчик Марк Винце написал отличную статью (blog.markvincze.com/how-to-use-envoy-as-a-load-balancer-in-kubernetes) о подготовке и настройке Envoy в Kubernetes.

Резюме

В Kubernetes все в конечном счете сводится к pod-оболочкам. Поэтому мы и вдавались в подробности: если вам кажется, что их было слишком много, приносим свои извинения. Вам не нужно понимать или помнить все, о чем мы рассказали в этой главе, — по крайней мере не сейчас. Позже вы можете столкнуться с проблемами, решить которые вам поможет более углубленный материал, рассмотренный здесь.

То, что следует запомнить.

- Метки — это пары вида «ключ — значение», которые идентифицируют ресурсы и могут использоваться в сочетании с селекторами для выбора определенных групп ресурсов.
- Pod-оболочки привлекаются к узлам, имеющим определенные атрибуты, и отталкиваются от узлов по аналогичному принципу: это называется *принадлежностью к узлам*. Например, вы можете сделать так, чтобы pod-оболочка могла работать только на узле в определенной зоне доступности.
- Если жесткая принадлежность к узлам может заблокировать работу pod-оболочки, то мягкая является скорее рекомендацией для планировщика. Вы можете сочетать несколько мягких правил принадлежности с разными весами.
- Принадлежность pod-оболочек выражает предпочтение одного `Pod`-объекта быть размещенным на том же самом узле, что и другой `Pod`-объект. Например, pod-оболочки, которые выигрывают от работы на одном узле, могут выразить это за счет взаимной принадлежности.
- Непринадлежность отталкивает pod-оболочки, а не привлекает их друг к другу. Например, непринадлежность реплик одного и того же `Pod`-объекта может помочь распределить эти реплики равномерно по кластеру.
- Ограничения — это способ маркировки узлов с помощью определенной информации, которая обычно касается их проблем или сбоев. Pod-оболочки по умолчанию не развертываются на ограниченных узлах.
- Допуски позволяют размещать pod-оболочку на узлах с определенными ограничениями. Вы можете использовать этот механизм для развертывания pod-оболочек только на выделенных узлах.
- `DaemonSet` позволяет разместить по одной копии pod-оболочки (например, агента ведения журнала) на каждом узле.
- `StatefulSet` запускает и останавливает реплики pod-оболочек в определенной последовательности, позволяя обращаться к ним по предсказуемым доменным именам с порядковыми номерами. Это идеально подходит для кластеризированных приложений, таких как базы данных.
- Запланированные задания выполняют pod-оболочку лишь один (или определенное количество) раз и затем завершаются. Задания Cronjob выполняют pod-оболочку периодически в заданные моменты времени.
- Механизм горизонтального автомасштабирования pod-оболочек следит за группами `Pod`-объектов и пытается оптимизировать заданный показатель (такой

как загруженность процессора). Для достижения этой цели он увеличивает или уменьшает количество реплик.

- Объекты `PodPreset` могут внедрять фрагменты общей конфигурации во все выбранные pod-оболочки в момент их создания. Например, с помощью `PodPreset` ко всем подходящим pod-оболочкам можно подключить определенный том.
- Определения пользовательских ресурсов (CRD) позволяют создавать собственные нестандартные объекты Kubernetes для хранения любых данных. Операторы — это клиентские программы для Kubernetes, которые реализуют оркестрацию для вашего конкретного приложения (например, MySQL).
- Ресурсы `Ingress` направляют запросы к разным сервисам, руководствуясь набором правил (например, сопоставляя подходящие участки URL-запроса). Они также могут разрывать TLS-соединения для ваших приложений.
- Istio — это инструмент, который предоставляет продвинутые сетевые возможности для микросервисов. Как любое приложение Kubernetes, он может быть установлен с помощью Helm.
- Envoy предоставляет механизм межсервисного взаимодействия и более гибкие возможности распределения нагрузки по сравнению со стандартными облачными балансировщиками.

10 Конфигурация и объекты Secret

> Если хочешь сохранить секрет, надо скрывать его и от себя.
>
> *Джордж Оруэлл. 1984*

Возможность отделить *логику* приложения Kubernetes от его *конфигурации* (то есть от любых значений или настроек, которые со временем могут поменяться) очень полезна. К конфигурационным значениям обычно относят параметры, предназначенные для определенной среды, DNS-адреса сторонних сервисов и учетные данные для аутентификации.

Конечно, все это можно поместить непосредственно в код, но такой подход недостаточно гибок. Например, для изменения конфигурационного значения тогда придется заново собирать и развертывать ваш код. Намного лучшим решением было бы отделить конфигурацию от кода и считывать ее из файла или переменных среды.

Kubernetes предоставляет несколько разных способов управления конфигурацией. Во-первых, вы можете передавать значения в приложение через переменные среды, указанные в спецификации pod-оболочки (см. подраздел «Переменные среды» на с. 192). Во-вторых, конфигурационные данные можно хранить непосредственно в Kubernetes, используя объекты `ConfigMap` и `Secret`.

В данной главе мы подробно исследуем эти объекты и рассмотрим некоторые практические подходы к управлению конфигурацией и конфиденциальными данными на примере демонстрационного приложения.

Объекты ConfigMap

`ConfigMap` — это основной объект для хранения конфигурационных данных в Kubernetes. Его можно представить в виде именованного набора пар «ключ — значение», в котором хранится конфигурация. С помощью `ConfigMap` вы можете предоставлять эти данные приложению, внедряя их в окружение pod-оболочки или создавая в ней соответствующий файл.

В этом разделе мы рассмотрим разные способы сохранения данных в `ConfigMap`, а также разберемся, как их оттуда доставать и подавать своим приложениям Kubernetes.

Создание ConfigMap

Представьте, что вам нужно создать в файловой системе своей pod-оболочки конфигурационный файл YAML с именем `config.yaml` и следующим содержимым:

```
autoSaveInterval: 60
batchSize: 128
protocols:
  - http
  - https
```

Как бы вы превратили этот набор значений в ресурс `ConfigMap`, который можно применить к Kubernetes?

Один из вариантов — записать данные в манифесте `ConfigMap` в том виде, в котором они указаны в YAML:

```
apiVersion: v1
data:
  config.yaml: |
    autoSaveInterval: 60
    batchSize: 128
    protocols:
      - http
      - https
kind: ConfigMap
metadata:
  name: demo-config
  namespace: demo
```

`ConfigMap` можно создать, написав манифест с нуля и добавив значения в раздел `data` файла `config.yaml`, как мы сделали в данном примере.

Но есть более простой способ: переложить эту работу на `kubectl` и создать ресурс напрямую из YAML-файла, как показано ниже:

```
kubectl create configmap demo-config --namespace=demo --from-file=config.yaml
configmap "demo-config" created
```

Чтобы экспортировать файл манифеста, соответствующий этому ресурсу `ConfigMap`, примените такую команду:

```
kubectl get configmap/demo-config --namespace=demo --export -o yaml
   >demo-config.yaml
```

В результате в файл `demo-config.yaml` будет записано представление ресурса кластера `ConfigMap` в формате YAML. Флаг `--export` удаляет метаданные, которые не следует держать в репозитории нашей инфраструктуры (см. подраздел «Экспорт ресурсов» на с. 165).

Задание переменных среды из ConfigMap

Итак, конфигурационные данные находятся в объекте `ConfigMap`. Как же теперь доставить их в контейнер? Давайте рассмотрим полноценный пример — наше демонстрационное приложение. Его код можно найти в папке `hello-config-env` репозитория `demo`.

Это то самое приложение, которое мы использовали в предыдущих главах для прослушивания HTTP-запросов и возврата ответов с приветствием (см. подраздел «Рассмотрение исходного кода» на с. 54).

Но на этот раз мы не станем размещать приветственную строку в коде приложения, а сделаем ее настраиваемой. Поэтому вы можете видеть небольшое изменение в функции `handler`, которая теперь считывает значение из переменной среды с именем `GREETING`:

```
func handler(w http.ResponseWriter, r *http.Request) {
        greeting := os.Getenv("GREETING")
        fmt.Fprintf(w, "%s, 世界\n", greeting)
}
```

Не обращайте внимания на отдельные особенности кода на Go, это всего лишь пример. Достаточно сказать, что в коде используется переменная среды `GREETING` при ответе на запросы, если она присутствует в момент его выполнения. К какому бы языку вы ни обратились для написания приложений, он, скорее всего, позволяет считывать переменные среды.

Теперь создадим объект `ConfigMap`, в котором будет храниться значение с приветствием. Файл манифеста объекта вместе с модифицированным приложением на Go можно найти в папке `helloc-onfig-env` репозитория `demo`.

Это выглядит так:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-config
data:
  greeting: Hola
```

Чтобы данные были доступны в среде контейнера, мы должны немного подправить наше развертывание. Вот та его часть, которая нас интересует:

```
spec:
  containers:
    - name: demo
      image: cloudnatived/demo:hello-config-env
      ports:
        - containerPort: 8888
      env:
        - name: GREETING
          valueFrom:
            configMapKeyRef:
              name: demo-config
              key: greeting
```

Обратите внимание на то, что здесь используется другой тег образа контейнера по сравнению с предыдущими примерами (см. подраздел «Идентификаторы образов» на с. 188). Тег `:hello-config-env` позволяет получить модифицированную версию демонстрационного приложения, которая считывает переменную `GREETING`: `cloudnatived/demo:hello-config-env`.

А вторым интересным моментом является раздел `env`. Как вы помните из подраздела «Переменные среды» на с. 192, переменные среды можно создавать из литералов, добавляя пары `name/value`.

У нас все еще есть `name`, но вместо `value` мы указали `valueFrom`. Это сообщает Kubernetes, что значение следует брать не из переменной-литерала, а откуда-то еще.

`configMapKeyRef` говорит о том, что нужно использовать определенный ключ в определенном объекте `ConfigMap`. Имя нужного нам объекта — `demo-config`, а ключ — `greeting`. Данные мы создали в манифесте `ConfigMap`, поэтому теперь они должны быть доступны для чтения в среде контейнера.

Если `ConfigMap` не существует, развертывание не сможет запуститься (его pod-оболочки будут иметь статус `CreateContainerConfigError`).

Это все, что нужно для работы обновленного приложения, поэтому можете развернуть полученные манифесты в своем кластере Kubernetes. Запустите следующую команду из папки репозитория `demo`:

```
kubectl apply -f hello-config-env/k8s/
configmap "demo-config" created
deployment.extensions "demo" created
```

Как и прежде, чтобы открыть приложение в своем веб-браузере, вам нужно перенаправить локальный порт к порту 8888 pod-оболочки:

```
kubectl port-forward deploy/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

(На сей раз мы не стали тратить время на создание сервиса: его следовало бы создать в настоящем промышленном приложении. В данном примере мы используем `kubectl`, чтобы перенаправить локальный порт прямо к развертыванию `demo`.)

Если все хорошо, то при открытии в браузере адреса http://localhost:9999/ вы должны увидеть следующее:

```
Hola, 世界
```

> **УПРАЖНЕНИЕ**
>
> Откройте другой терминал (команда `kubectl port-forward` должна продолжать работать) и отредактируйте файл `configmap.yaml`, изменив приветствие. Примените его заново с помощью `kubectl`. Обновите веб-браузер. Изменился ли результат? Если нет, то почему? Что нужно сделать, чтобы приложение прочитало обновленное значение? Разобраться в этом вам поможет подраздел «Обновление pod-оболочек при изменении конфигурации» на с. 243.

Установка всей среды из ConfigMap

Вы можете установить одну или две переменные среды из отдельных ключей `ConfigMap`, но, как мы видели в предыдущем примере, при большом количестве значений это может быть утомительным занятием.

К счастью, раздел envFrom позволяет легко взять все ключи из ConfigMap и превратить их в переменные среды:

```
spec:
  containers:
    - name: demo
      image: cloudnatived/demo:hello-config-env
      ports:
        - containerPort: 8888
      envFrom:
        - configMapRef:
            name: demo-config
```

Теперь каждый параметр в объекте demo-config станет переменной в среде контейнера. Поскольку в нашем примере ConfigMap ключ называется greeting, переменная среды будет иметь такое же имя (в нижнем регистре). Чтобы при использовании envFrom перевести свои переменные в верхний регистр, измените их внутри ConfigMap.

Вы также можете устанавливать переменные среды для контейнера обычными способами, используя env и помещая литералы в файл манифеста или применяя ConfigMapKeyRef, как в нашем предыдущем примере. Kubernetes позволяет использовать env и envFrom одновременно.

Если две переменные — одна в env, а другая в envFrom — имеют одно и то же имя, первая получает приоритет. Например, при установке переменной GREETING в env и в объекте ConfigMap, на который ссылается envFrom, значение из env перезапишет взятое из ConfigMap.

Использование переменных среды в аргументах командной строки

Возможность размещения конфигурационных данных в среде контейнера имеет свои плюсы, но иногда эти данные необходимо предоставить конечной точке контейнера в виде аргументов командной строки.

Для этого, как и в предыдущем примере, можно взять переменные среды из ConfigMap. При этом для доступа к ним в командной строке используется специальный синтаксис Kubernetes вида $(VARIABLE).

Данный пример можно найти в файле deployment.yaml, размещенном в папке hello-config-args репозитория demo:

```
spec:
  containers:
```

```
    - name: demo
      image: cloudnatived/demo:hello-config-args
      args:
        - "-greeting"
        - "$(GREETING)"
      ports:
        - containerPort: 8888
      env:
        - name: GREETING
          valueFrom:
            configMapKeyRef:
              name: demo-config
              key: greeting
```

Здесь в спецификацию контейнера мы добавили поле `args`, которое передаст наши пользовательские аргументы точке входа по умолчанию (`/bin/demo`).

Все, что выглядит в манифесте как `$(VARIABLE)`, Kubernetes заменяет значением переменной среды `VARIABLE`. Поскольку мы создали переменную `GREETING` и задали ее значение из `ConfigMap`, она теперь доступна для использования в командной строке контейнера.

В результате применения этих манифестов значение `GREETING` будет передано приложению `demo` следующим образом:

```
kubectl apply -f hello-config-args/k8s/
configmap "demo-config" configured
deployment.extensions "demo" configured
```

Результат должен быть виден в вашем браузере:

```
Salut, 世界
```

Создание конфигурационных файлов из объектов ConfigMap

Мы уже познакомились с двумя разными способами доставки данных из объектов `ConfigMap` в приложения Kubernetes: через окружение и командную строку контейнера. Однако более сложные программы часто предполагают считывание их конфигурации из файла на диске.

К счастью, Kubernetes позволяет создавать такие файлы непосредственно из `ConfigMap`. Для начала изменим наш ресурс `ConfigMap` таким образом, чтобы вместо одного ключа он хранил целый YAML-файл, содержащий в данном примере один ключ, но на самом деле их там может быть и целая сотня:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-config
data:
  config: |
    greeting: Buongiorno
```

Вместо того чтобы устанавливать ключ `greeting`, как в предыдущем примере, мы создадим новый ключ под названием `config` и припишем ему *блок* данных (вертикальная черта | в YAML говорит о том, что за ней идет блок необработанных данных). Данные такие:

```
greeting: Buongiorno
```

Это соответствует формату YAML, но имейте в виду, что возможен и любой другой формат, включая JSON, TOML или обычный текст. Как бы то ни было, Kubernetes в итоге запишет весь блок данных в его оригинальном виде в файл внутри контейнера.

Итак, мы сохранили необходимые данные. Теперь развернем их в Kubernetes. В папке `hello-config-file` репозитория `demo` вы найдете шаблон развертывания следующего содержания:

```
spec:
  containers:
    - name: demo
      image: cloudnatived/demo:hello-config-file
      ports:
        - containerPort: 8888
      volumeMounts:
      - mountPath: /config/
        name: demo-config-volume
        readOnly: true
  volumes:
  - name: demo-config-volume
    configMap:
      name: demo-config
      items:
      - key: config
        path: demo.yaml
```

Глядя на раздел контейнера `volumes`, вы можете заметить, что мы создаем том `demo-config-volume` из существующего объекта `ConfigMap` с именем `demo-config`.

В разделе `volumeMounts` контейнера мы подключаем этот том в поле `mountPath: /config/`, выбираем ключ `config` и записываем его в `demo.yaml`. В результате

Kubernetes создаст в контейнере файл `/config/demo.yaml`, содержащий данные `demo-config` в формате YAML:

```
greeting: Buongiorno
```

Демонстрационное приложение прочитает эту конфигурацию из файла в момент запуска. Как и прежде, примените манифесты с помощью такой команды:

```
kubectl apply -f hello-config-file/k8s/
configmap "demo-config" configured
deployment.extensions "demo" configured
```

Результат должен быть виден в вашем браузере:

```
Buongiorno, 世界
```

Если вы хотите узнать, как выглядят данные `ConfigMap` в кластере, выполните следующую команду:

```
kubectl describe configmap/demo-config
Name:         demo-config
Namespace:    default
Labels:       <none>
Annotations:
kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"v1",
"data":{"config":"greeting: Buongiorno\n"},"kind":"ConfigMap","metadata":
{"annotations":{},"name":"demo-config","namespace":"default...

Data
====
config:
greeting: Buongiorno

Events: <none>
```

Если обновить объект `ConfigMap` и изменить его значения, соответствующий файл (в нашем примере это `/config/demo.yaml`) будет обновлен автоматически. Некоторые (но не все) приложения способны сами обнаружить, что файл изменился, и заново его прочитать.

Чтобы изменения вступили в силу, приложение можно развернуть повторно (см. подраздел «Обновление pod-оболочек при изменении конфигурации» на с. 243). Если же оно способно инициировать динамическое обновление — например, по сигналу Unix (такому как `SIGHUP`) или путем выполнения команды в контейнере — это может не понадобиться.

Обновление pod-оболочек при изменении конфигурации

Представьте, что в вашем кластере есть развертывание и вы хотите поменять некоторые значения в его `ConfigMap`. Если вы используете чарт Helm (см. раздел «Helm: диспетчер пакетов для Kubernetes» на с. 102), обнаружить изменение конфигурации и перезагрузить ваши pod-оболочки можно автоматически с помощью одного изящного приема. Добавьте следующую аннотацию в спецификацию своего развертывания:

```
checksum/config: {{ include (print $.Template.BasePath "/configmap.yaml") .
    | sha256sum }}
```

Теперь шаблон развертывания содержит контрольную сумму конфигурационных параметров: при изменении параметров сумма обновится. Если выполнить команду `helm upgrade`, Helm обнаружит, что спецификация развертывания изменилась, и перезапустит все pod-оболочки.

Конфиденциальные данные в Kubernetes

Мы уже знаем, что объект `ConfigMap` предоставляет гибкий механизм хранения и доступа к конфигурационным данным в кластере. Однако у большинства приложений есть информация, которая является секретной и конфиденциальной: например, пароли или API-ключи. Ее можно хранить и в `ConfigMap`, но такое решение неидеально.

Вместо этого Kubernetes предлагает объект специального типа, предназначенный для хранения конфиденциальных данных: `Secret`. Далее рассмотрим на примере, как данный объект можно применить в нашем демонстрационном приложении.

Для начала взгляните на манифест Kubernetes для объекта `Secret` (см. `hello-secret-env/k8s/secret.yaml`):

```
apiVersion: v1
kind: Secret
metadata:
  name: demo-secret
stringData:
  magicWord: xyzzy
```

В этом примере закрытый ключ `magicWord` имеет значение `xyzzy` (en.wikipedia.org/wiki/Xyzzy_(computing)). Слово `xyzzy` вообще очень полезное в мире компьютеров. По аналогии с `ConfigMap` в объекте `Secret` можно размещать множество ключей и значений. Здесь для простоты мы используем лишь одну пару «ключ — значение».

Использование объектов Secret в качестве переменных среды

Как и `ConfigMap`, объект `Secret` можно сделать доступным в контейнере в виде переменных среды или файла на его диске. В следующем примере мы присвоим переменной среды значение из `Secret`:

```
spec:
  containers:
    - name: demo
      image: cloudnatived/demo:hello-secret-env
      ports:
        - containerPort: 8888
      env:
        - name: GREETING
          valueFrom:
          secretKeyRef:
            name: demo-secret
            key: magicWord
```

Мы установили переменную `GREETING` точно так же, как это было сделано в случае с `ConfigMap`, только теперь это `secretKeyRef` вместо `configMapKeyRef` (см. подраздел «Задание переменных среды из ConfigMap» на с. 236).

Выполните следующую команду в репозитории `demo`, чтобы применить манифесты:

```
kubectl apply -f hello-secret-env/k8s/
deployment.extensions "demo" configured
secret "demo-secret" created
```

Как и раньше, перенаправьте локальный порт к развертыванию, чтобы увидеть результат в своем браузере:

```
kubectl port-forward deploy/demo 9999:8888
Forwarding from 127.0.0.1:9999 -> 8888
Forwarding from [::1]:9999 -> 8888
```

При открытии адреса http://localhost:9999/ вы должны увидеть следующее:

```
The magic word is "xyzzy"
```

Запись объектов Secret в файлы

В этом примере мы подключим объект `Secret` к контейнеру в виде файла. Код находится в папке `hello-secret-file` репозитория `demo`.

Чтобы подключить `Secret` в виде файла, воспользуемся следующим развертыванием:

```
spec:
  containers:
    - name: demo
      image: cloudnatived/demo:hello-secret-file
      ports:
        - containerPort: 8888
      volumeMounts:
        - name: demo-secret-volume
          mountPath: "/secrets/"
          readOnly: true
  volumes:
    - name: demo-secret-volume
      secret:
        secretName: demo-secret
```

Как и в подразделе «Создание конфигурационных файлов из объектов ConfigMap» на с. 240, мы создаем том (в данном случае это `demo-secret-volume`) и подключаем его к контейнеру в разделе спецификации `volumeMounts`. В поле `mountPath` указано `/secrets`, поэтому Kubernetes создаст в этой папке по одному файлу для каждой пары «ключ — значение», определенной в объекте `Secret`.

В нашем примере мы определили только одну пару «ключ — значение» с именем `magicWord`, поэтому манифест создаст в контейнере один файл `/secrets/magicWord` с конфиденциальными данными, доступный исключительно для чтения.

Если применить этот манифест таким же образом, как и в предыдущем примере, должен получиться тот же результат:

```
The magic word is "xyzzy"
```

Чтение объектов Secret

В предыдущем разделе мы использовали команду `kubectl describe` для вывода содержимого `ConfigMap`. Можно ли то же самое сделать с `Secret`?

```
kubectl describe secret/demo-secret
Name:         demo-secret
```

```
Namespace:    default
Labels:       <none>
Annotations:
Type:         Opaque

Data
====
magicWord:  5 bytes
```

Обратите внимание на то, что сами данные не отображаются. Объекты `Secret` в Kubernetes имеют тип `Opaque`: это означает, что их содержимое не показывается в выводе `kubectl describe`, журнальных записях и терминале, благодаря чему невозможно случайно раскрыть конфиденциальную информацию.

Чтобы просмотреть закодированную версию конфиденциальных данных в формате YAML, воспользуйтесь командой `kubectl get`:

```
kubectl get secret/demo-secret -o yaml
apiVersion: v1
data:
  magicWord: eHl6enk=
kind: Secret
metadata:
...
type: Opaque
```

base64

Что за `eHl6enk=`, совсем не похожий на наше исходное значение? На самом деле это объект `Secret`, представленный в кодировке *base64*. Base64 — это схема кодирования произвольных двоичных данных в виде строки символов.

Поскольку конфиденциальная информация может быть двоичной и недоступной для вывода (как, например, в случае с ключом шифрования TLS), объекты `Secret` всегда хранятся в формате base64.

Текст `beHl6enk=` является версией нашего секретного слова `xyzzy`, закодированной в base64. В этом можно убедиться, если выполнить в терминале команду `base64 --decode`:

```
echo "eHl6enk=" | base64 --decode
xyzzy
```

Таким образом, несмотря на то, что Kubernetes защищает вас от случайного вывода конфиденциальных данных в терминале или журнальных файлах, при наличии

прав на чтение объектов `Secret` в определенном пространстве имен эти данные можно получить в формате base64 и впоследствии их раскодировать.

Если вам нужно закодировать в base64 какой-нибудь текст (например, чтобы поместить его в `Secret`), используйте команду `base64` без аргументов:

```
echo xyzzy | base64
eHl6enkK
```

Доступ к объектам Secret

Кто может читать и редактировать объекты `Secret`? Это определяется RBAC — механизмом контроля доступа (подробно его обсудим в подразделе «Введение в управление доступом на основе ролей» на с. 258). Если вы используете кластер, в котором система RBAC отсутствует или не включена, все ваши объекты `Secret` доступны любым пользователям и контейнерам (позже мы объясним, что у вас не должно быть ни одного промышленного кластера без RBAC).

Пассивное шифрование данных

А что насчет тех, кто имеет доступ к базе данных *etcd*, в которой Kubernetes хранит всю свою информацию? Могут ли они прочитать конфиденциальные данные, не имея прав на чтение объектов `Secret` через API?

Начиная с версии 1.7, Kubernetes поддерживает *пассивное шифрование данных*. Это означает, что конфиденциальная информация внутри *etcd* хранится на диске в зашифрованном виде и не может быть прочитана даже тем, кто имеет прямой доступ к базе данных. Для ее расшифровки нужен ключ, который есть только у сервера API Kubernetes. В правильно сконфигурированном кластере пассивное шифрование должно быть включено.

Проверить, работает ли пассивное шифрование в вашем кластере, можно таким образом:

```
kubectl describe pod -n kube-system -l component=kube-apiserver |grep encryption
     --experimental-encryption-provider-config=...
```

Если вы не видите флага `experimental-encryption-provider-config`, пассивное шифрование не включено. При использовании Google Kubernetes Engine или других сервисов по управлению Kubernetes ваши данные шифруются с помощью иного механизма, поэтому флаг будет отсутствовать. Узнайте у своего поставщика Kubernetes, шифруется ли содержимое `etcd`.

Хранение конфиденциальных данных

Есть такие ресурсы Kubernetes, которые никогда не следует удалять из кластера: например, особо важные объекты `Secret`. Вы можете уберечь ресурс от удаления с помощью аннотации, предоставляемой диспетчером Helm:

```
kind: Secret
metadata:
  annotations:
    "helm.sh/resource-policy": keep
```

Стратегии управления объектами Secret

В примере из предыдущего раздела конфиденциальные данные защищались от несанкционированного доступа сразу после сохранения в кластере. Но в файлах манифестов они хранились в виде обычного текста.

Вы никогда не должны размещать конфиденциальную информацию в файлах, которые находятся в системе контроля версий. Как же безопасно администрировать и хранить такую информацию до того, как применить ее к кластеру Kubernetes?

Вы можете выбрать любые инструменты или стратегии для работы с конфиденциальными данными в своих приложениях, но все равно вам понадобится ответить как минимум на следующие вопросы.

- Где хранить конфиденциальные данные, чтобы они были высокодоступными?
- Как сделать конфиденциальные данные доступными для ваших активных приложений?
- Что должно происходить с вашими приложениями, когда вы заменяете или редактируете конфиденциальные данные?

В этой главе мы рассмотрим три самые популярные стратегии работы с конфиденциальной информацией и то, каким образом каждая из них решает эти задачи.

Шифрование конфиденциальных данных в системе контроля версий

Первый вариант — хранить конфиденциальную информацию в зашифрованном виде прямо в коде внутри репозитория системы контроля версий, расшифровывая ее на этапе развертывания.

Это, наверное, самый простой вариант. Конфиденциальные данные находятся непосредственно в репозиториях исходного кода, но не в открытом виде. Они шифруются таким образом, чтобы расшифровать их можно было только с помощью определенного доверенного ключа.

При развертывании приложения данные расшифровываются прямо перед применением манифестов Kubernetes к кластеру. Дальше приложение может читать их, как любые другие конфигурационные сведения.

Шифрование конфиденциальной информации в системе контроля версий позволяет просматривать и отслеживать изменения, которые в нее вносятся, — точно так же, как и при работе с кодом приложения. На нее распространяется и высокодоступность ваших репозиториев.

Чтобы отредактировать или заменить данные, сначала их нужно расшифровать в вашей локальной копии исходных текстов, обновить и зафиксировать изменения в системе контроля версий.

Эта стратегия проста в реализации и не имеет никаких зависимостей, кроме ключа и инструмента шифрования/дешифрования (см. раздел «Шифрование конфиденциальных данных с помощью Sops» на с. 252), однако у нее есть один потенциальный недостаток: если конфиденциальная информация используется несколькими приложениями, ее придется скопировать в исходный код каждого из них. Это означает, что замена данных потребует дополнительных усилий, так как вы должны будете убедиться в том, что нашли и заменили их во всех приложениях.

Существует также серьезный риск нечаянно зафиксировать в системе контроля версий конфиденциальные данные в открытом виде — ошибки ведь случаются. И даже если подобное произойдет в закрытом репозитории, любые зафиксированные таким образом данные должны считаться скомпрометированными: их нужно будет заменить как можно быстрее. Возможно, доступ к ключу лучше выдавать отдельным лицам, а не всем разработчикам.

Тем не менее использование стратегии *шифрования конфиденциальной информации в исходном коде* будет хорошей отправной точкой для небольших организаций с не самыми важными данными. Такой подход требует относительно небольших усилий, легко внедряется, и благодаря своей гибкости может применяться в нескольких приложениях и для разных типов конфиденциальных данных. В последнем разделе этой главы мы рассмотрим некоторые инструменты для шифрования/дешифрования в исходном коде, но сначала вкратце опишем альтернативные стратегии.

Удаленное хранение конфиденциальных данных

Еще одним вариантом управления конфиденциальной информацией является ее размещение в файле (или нескольких файлах) удаленного защищенного хранилища, такого как бакет AWS S3 или Google Cloud Storage. При развертывании отдельного приложения ему будут предоставлены предварительно загруженные и расшифрованные файлы. Это схоже с предыдущей стратегией, только конфиденциальные данные находятся не в репозитории исходного кода, а в центральном хранилище. Обеим стратегиям подойдут одни и те же инструменты для шифрования/дешифрования.

Такой подход решает проблему дублирования данных в разных репозиториях, но требует более тщательного проектирования и координации: ведь важно, чтобы при развертывании применялась нужная нам информация. Можно также использовать отдельный инструмент для управления конфиденциальными данными — со всеми его преимуществами и без необходимости установки и администрирования дополнительного программного компонента или рефакторинга вашего приложения для взаимодействия с ним.

Поскольку ваша конфиденциальная информация находится вне системы контроля версий, вам придется выработать процесс ее упорядоченного обновления, желательно с ведением журнала аудита (кто, что, когда и зачем изменил) и процедурой, аналогичной рассмотрению и одобрению запросов на внесение изменений.

Использование специального инструмента для управления конфиденциальными данными

Две предыдущие стратегии подходят для большинства организаций, но если речь идет об очень крупных масштабах, лучше задуматься об использовании специального продукта для управления конфиденциальными данными, например Vault от Hashicorp, Keywhiz от Square, AWS Secrets Manager или Key Vault от Azure. Эти инструменты берут на себя безопасное хранение всех подобных данных вашего приложения в одном центральном и высокодоступном месте и позволяют следить за тем, какие пользователи и служебные учетные записи имеют право их добавлять, удалять, изменять или просматривать.

В системе управления конфиденциальными данными все действия записываются и могут быть просмотрены, что упрощает анализ дыр в безопасности, а также помогает убедиться в выполнении корпоративных стандартов. Некоторые из этих

продуктов позволяют автоматически заменять секретные ключи на регулярной основе — что и само по себе является хорошей идеей, но также может потребоваться в рамках разнообразных корпоративных политик.

Каким образом эти инструменты позволяют приложениям получать конфиденциальные данные? Один из распространенных подходов состоит в использовании служебной учетной записи с доступом на чтение защищенного хранилища: приложение читает только ту информацию, которая ему нужна. У разработчиков могут быть свои собственные учетные данные с правами на чтение и запись только в тех программах, с которыми они работают.

Центральная система управления конфиденциальными данными является наиболее мощной и гибкой опцией, доступной на сегодняшний день, но вместе с тем она существенно усложняет инфраструктуру. Помимо настройки и обслуживания защищенного хранилища, вам придется добавлять соответствующий инструментарий или промежуточный слой во все приложения и сервисы, использующие его содержимое. Конечно, вы можете изменить или перепроектировать код таким образом, чтобы он обращался к защищенному хранилищу напрямую, но это может оказаться более затратным и кропотливым процессом, чем просто разместить перед хранилищем прослойку, которая извлекает конфиденциальные данные и помещает их в среду приложения или конфигурационный файл.

Среди разных продуктов одним из наиболее популярных является Vault от Hashicorp.

Рекомендации

На первый взгляд может показаться, что выбор специальных систем управления конфиденциальными данными вроде Vault достаточно логичен, но начинать с них мы не советуем: попробуйте облегченные инструменты, такие как Sops (см. раздел «Шифрование конфиденциальных данных с помощью Sops» на с. 252), которые производят шифрование прямо в вашем исходном коде.

Почему? Хотя бы потому, что у вас может быть не так уж и много конфиденциальной информации. Если вы не работаете с очень сложной и переплетенной инфраструктурой (чего в любом случае следует избегать), любому отдельному приложению, скорее всего, требуется один-два экземпляра секретных данных: например, API-ключи и токены для других сервисов или же средства доступа к базе данных. Если какое-то отдельно взятое приложение нуждается в большом количестве подобной информации, подумайте об альтернативном варианте — разместите ее в едином зашифрованном файле.

Как и раньше, когда мы в этой книге говорили о проблемах, так и в случае с управлением конфиденциальными данными мы предпочитаем прагматичный подход. Если простая и легкая в использовании система решает ваши задачи, начните с нее: позже вы всегда сможете перейти на более мощную или сложную конфигурацию. Часто на начальных стадиях проекта невозможно предугадать, сколько именно конфиденциальной информации будет использоваться. А пока вы не уверены, выбирайте тот вариант, который позволит приступить к работе быстрее всего. Но будьте готовы поменять свой выбор в будущем.

Тем не менее, если вам с самого начала известно о нормативных или прочих ограничениях относительно работы с конфиденциальной информацией, лучше учтите это в ходе проектирования. Скорее всего, вам придется посмотреть в сторону специальных решений.

Шифрование конфиденциальных данных с помощью Sops

Допустим, вы решили заниматься шифрованием самостоятельно, по крайней мере на первых порах. Вам понадобится инструмент, способный работать с вашим исходным кодом и файлами данных. Sops (от англ. secrets operations — операции с конфиденциальными данными) от проекта Mozilla — это система шифрования/дешифрования, которая поддерживает YAML, JSON и двоичные файлы. Она предлагает несколько механизмов шифрования, включая PGP/GnuPG, Azure Key Vault, Key Management Service (KMS) от AWS и Cloud KMS от Google.

Знакомство с Sops

Для ознакомления с инструментом Sops продемонстрируем, что он делает. Вместо того чтобы шифровать весь файл, Sops шифрует только отдельные конфиденциальные значения. Если в вашем файле находится такой текст:

```
password: foo
```

после его шифрования с помощью Sops вы получите файл следующего содержания:

```
password: ENC[AES256_GCM,data:p673w==,iv:YY=,aad:UQ=,tag:A=]
```

Это упрощает редактирование и просмотр кода, особенно в запросах на включение внесенных изменений. Причем вам не нужно расшифровывать данные, чтобы понять, на что вы смотрите.

Инструкции по установке и использованию Sops можно найти на домашней странице проекта (github.com/mozilla/sops).

В остальной части этой главы мы рассмотрим примеры работы с Sops, расскажем, как этот инструмент работает с Kubernetes, и добавим несколько конфиденциальных значений из хранилища Sops в наше демонстрационное приложение. Но сначала следует упомянуть о существовании альтернативных инструментов для шифрования секретных данных. Если вы уже используете один из них, ничего страшного: главное, чтобы он умел шифровать и расшифровывать значения внутри текстовых файлов, как это делает Sops.

Мы фанаты Helm, как вы знаете. Если вы дочитали до этого места и если вам необходимо работать с конфиденциальной информацией в чартах Helm, можете применять для этого Sops в сочетании с дополнением `helm-secrets`. При выполнении команд `helm upgrade` или `helm install` дополнение расшифрует ваши секретные значения для развертывания. Больше информации о Sops, включая инструкции по установке и использованию, можно найти в репозитории на GitHub (github.com/futuresimple/helm-secrets).

Шифрование файла с помощью Sops

Попробуем воспользоваться Sops и зашифруем файл. Мы уже упоминали, что Sops не занимается шифрованием как таковым и делегирует его внутреннему инструменту GnuPG (популярная реализация протокола PGP [pretty good privacy — «довольно хорошая конфиденциальность»] с открытым исходным кодом). В этом примере мы воспользуемся Sops и GnuPG для шифрования файла с конфиденциальными данными. В итоге у нас получится файл, который можно будет безопасно зафиксировать в системе контроля версий.

Мы не станем вдаваться в подробности работы шифрования PGP — просто имейте в виду, что это криптографическая система с *открытым ключом*, так же как SSH и TLS. Для кодирования данных вместо одного используется пара ключей: открытый и закрытый. Открытым ключом можно свободно делиться с другими пользователями, но закрытый всегда должен оставаться только у вас.

Попробуем сгенерировать вашу пару ключей. Для начала установите GnuPG (gnupg.org/download), если еще не сделали этого.

Закончив с установкой, сгенерируйте пару ключей с помощью такой команды:

```
gpg --gen-key
```

Когда операция успешно завершится, обратите внимание на строчку `Key fingerprint` (с последовательностью шестнадцатеричных цифр): она однозначно идентифицирует ваш ключ, и вы будете использовать ее на следующем этапе.

Итак, вы обзавелись парой ключей. Теперь применим ее в сочетании с Sops, чтобы зашифровать файл. Для этого на вашем компьютере должен быть установлен пакет Sops, доступный для загрузки (github.com/mozilla/sops/releases) или установки с помощью Go:

```
go get -u go.mozilla.org/sops/cmd/sops
sops -v
sops 3.0.5 (latest)
```

Теперь создадим тестовый конфиденциальный файл, который будет шифроваться:

```
echo "password: secret123" > test.yaml
cat test.yaml
password: secret123
```

И наконец, зашифруем его с помощью Sops. Передайте слепок своего ключа переключателю `--pgp`, предварительно убрав из него все пробелы:

```
sops --encrypt --in-place --pgp E0A9AF924D5A0C123F32108EAF3AA2B4935EA0AB
test.yaml cat test.yaml
password: ENC[AES256_GCM,data:Ny220Ml8JoqP,iv:HMkwA8eFFmdUU1Dle6NTpVgy8vlQu/
6Zqx95Cd/+NL4=,tag:Udg9Wef8coZRbPb0foOOSA==,type:str]
sops:
  ...
```

Получилось! Файл `test.yaml` надежно зашифрован, и значение `password` можно расшифровать только с помощью вашего закрытого ключа. Наверное, также вы заметили, что программа Sops добавила в конец файла некие метаданные, чтобы знать, как его расшифровывать в будущем.

Еще одним удобным свойством Sops является то, что шифруется только *значение*, поэтому файл сохраняет формат YAML (вы можете видеть, что зашифрованные данные помечены как `password`). Если в вашем YAML-файле содержится длинный список пар «ключ — значение», ключи останутся в исходном виде.

Желая убедиться в том, что можете получить обратно ваши зашифрованные данные и что они за это время не изменились, выполните такую команду:

```
sops --decrypt test.yaml
You need a passphrase to unlock the secret key for
user: "Justin Domingus <justin@example.com>"
2048-bit RSA key, ID 8200750F, created 2018-07-27 (main key ID 935EA0AB)
Enter passphrase: *секретная фраза*

password: secret123
```

Помните секретную фразу, которую вы выбрали, когда генерировали свою пару ключей? Надеемся, что да, потому что сейчас ее нужно ввести! Если вы запомнили все правильно, на экране должно появиться расшифрованное значение `password: secret123`.

Теперь, научившись использовать Sops, вы можете зашифровать любые конфиденциальные данные в своем исходном коде, будь то конфигурационные файлы приложения, ресурсы Kubernetes в формате YAML или что-то другое.

Когда придет время развертывать ваше приложение, используйте Sops в режиме дешифрования, чтобы сгенерировать нужные вам конфиденциальные данные в виде обычного текста (но не забудьте их удалить и не фиксируйте их в системе контроля версий!).

В следующей главе мы покажем, как подобным же образом использовать Sops вместе с чартами Helm. Helm позволяет не только расшифровывать конфиденциальную информацию при развертывании вашего приложения, но и применять разные секретные значения в зависимости от среды развертывания: например, одни для `staging` и другие для `production` (см. подраздел «Управление конфиденциальными данными чартов Helm с помощью Sops» на с. 289).

Использование внутреннего механизма KMS

Если вы используете Amazon KMS или Google Cloud KMS для управления ключами в облаке, можете делать это в связке с Sops. Работа с ключами KMS происходит точно так же, как в нашем примере с PGP, только метаданные в файле будут другими. Раздел `sops:` внизу может выглядеть следующим образом:

```
sops:
  kms:
    - created_at: 1441570389.775376
      enc: CiC....Pm1Hm
      arn: arn:aws:kms:us-east-1:656532927350:key/920aff2e...
```

Точно так же, как и в примере с PGP, идентификатор ключа (`arn:aws:kms...`) встраивается в файл, чтобы позже программа Sops знала, как его расшифровывать.

Резюме

Конфигурация и конфиденциальные данные — это одна из тех тем, которыми люди интересуются больше всего в контексте Kubernetes. Мы рады возможности посвятить ей целую главу и описать методы подключения приложений с нужными настройками и данными.

Вот наиболее важное, что вам следует знать.

- Отделяйте ваши конфигурационные данные от программного кода и развертывайте их с помощью таких объектов Kubernetes, как `ConfigMap` и `Secret`. Тогда вам не придется заново развертывать приложение при каждом изменении пароля.
- Чтобы поместить данные в `ConfigMap`, их можно записать непосредственно в файл манифеста Kubernetes или преобразовать имеющийся файл в формате YAML в спецификацию `ConfigMap` с помощью `kubectl`.
- Когда данные находятся в `ConfigMap`, вы можете внедрить их в среду контейнера или в аргументы командной строки его точки входа. Как вариант, их можно записать в файл, который подключается к контейнеру.
- Объекты `Secret` работают по аналогии с `ConfigMap`, только данные шифруются в пассивном режиме и отображаются в закодированном виде в выводе `kubectl`.
- Для простой и гибкой работы с конфиденциальными данными их можно хранить прямо в репозитории исходного кода, шифруя с использованием Sops или другого инструмента для шифрования текста.
- Не следует изобретать велосипед для управления конфиденциальной информацией, особенно на первых порах: начните с чего-нибудь простого, что разработчикам можно будет легко настроить.
- Если конфиденциальные данные используются сразу несколькими приложениями, вы можете хранить их в облачном бакете (в зашифрованном виде) и извлекать оттуда на этапе развертывания.
- Для работы с конфиденциальной информацией на уровне предприятия вам понадобится специальный сервис, такой как Vault. Однако с его использованием лучше немного повременить, пока вы не будете уверены в том, что он вам действительно нужен.
- Sops — это криптографический инструмент, который работает с файлами вида «ключ — значение», такими как YAML или JSON. Он может брать ключи шифрования из локального хранилища GnuPG или из облачных сервисов для управления ключами, таких как Amazon KMS или Google Cloud KMS.

11 Безопасность и резервное копирование

> Если вы думаете, что технологии могут решить проблемы с безопасностью, у вас нет понимания ни этих проблем, ни технологий.
>
> *Брюс Шнайер. Прикладная криптография*

В этой главе мы исследуем безопасность и механизмы управления доступом в Kubernetes, в том числе и на основе ролей (Role-Based Access Control, или RBAC), рассмотрим некоторые инструменты и сервисы для сканирования уязвимостей и покажем, как выполнять резервное копирование данных и состояния кластера (что еще важнее, как их восстанавливать). Мы также обратим внимание на некоторые важные методики получения информации о том, что происходит в вашей системе Kubernetes.

Управление доступом и права доступа

Небольшие технологические компании обычно начинаются всего с нескольких сотрудников, у каждого из которых есть полный доступ ко всем системам.

Но по мере разрастания организации в какой-то момент выдача прав администратора всем подряд перестает быть хорошей идеей: высока вероятность, что кто-то допустит ошибку и изменит то, чего менять нельзя. Это касается и Kubernetes.

Управление доступом в кластере

Один из самых простых и эффективных способов обезопасить свой кластер Kubernetes — ограничить доступ к нему. К кластерам Kubernetes обычно обращаются две категории людей: *администраторы* и *разработчики*, и их должностные обязанности часто требуют разных прав доступа и привилегий.

Кроме того, у вас может быть несколько сред развертывания: например, промышленная и тестовая. Для них потребуются отдельные политики, в зависимости от требований вашей организации. Доступ к промышленной среде дается лишь нескольким лицам, тогда как тестовая среда открывается для более широкого круга инженеров.

Как мы уже видели в пункте «Нужно ли мне больше одного кластера?» на с. 140, для промышленного выполнения и для заключительного и обычного тестирования в большинстве случаев лучше иметь отдельные кластеры. Тогда, если кто-то развернет в среде заключительного тестирования код, нарушающий работу узлов кластера, это не повлияет на промышленную систему.

В ситуации, когда одна команда не должна иметь доступ к процессу разработки и развертывания другой, каждой из них можно выделить по кластеру с разными учетными данными.

Это, безусловно, наиболее безопасный подход, но использование дополнительных кластеров имеет свои недостатки. Каждый из них придется обновлять и мониторить, к тому же множество мелких кластеров обычно работают не так эффективно, как один большой.

Введение в управление доступом на основе ролей

Другой способ управлять доступом — контролировать список тех, кто может выполнять определенные операции внутри кластера. Для этого Kubernetes предлагает систему управления доступом на основе ролей (Role-Based Access Control, или RBAC).

Система RBAC предназначена для выдачи определенных прав доступа определенным пользователям (или служебным учетным записям, относящимся к автоматизированным системам). Например, вы можете позволить просматривать все pod-оболочки в кластере тому пользователю, которому это необходимо.

Первое и самое важное, что нужно знать о системе RBAC, это то, что она должна быть включена. В Kubernetes 1.6 система появилась в виде параметра при настройке кластера. Но включена ли она в кластере, зависит от вашего облачного провайдера или установщика Kubernetes.

Если вы самостоятельно размещаете свой кластер, попробуйте следующую команду (она покажет, включена ли у вас поддержка RBAC):

```
kubectl describe pod -n kube-system -l component=kube-apiserver
Name:         kube-apiserver-docker-for-desktop
Namespace:    kube-system
...
Containers:
  kube-apiserver:
      ...
    Command:
      kube-apiserver
      ...
      --authorization-mode=Node,RBAC
```

Если флаг `--authorization-mode` не содержит значение `RBAC`, значит, RBAC в вашем кластере не работает. Для получения информации, как пересобрать свой кластер так, чтобы он поддерживал RBAC, сверьтесь с документацией своего поставщика сервисов или установщика.

Без RBAC любой, кто имеет доступ к кластеру, может делать что угодно: может даже запустить произвольный код или удалить рабочие задания. Вряд ли это то, что вам нужно.

Понимание ролей

Итак, предположим, что система RBAC включена. Как же она работает? Самые важные понятия, которые следует знать, — это пользователи, роли и привязки.

Всякий раз, подключаясь к Kubernetes, вы делаете это от имени определенного пользователя. Сам процесс аутентификации зависит от вашего провайдера: в Google Kubernetes Engine, например, для получения токена доступа для того или иного кластера используется инструмент `gcloud`.

В кластере присутствуют и другие пользователи: в каждом пространстве имен, например, есть служебная учетная запись по умолчанию. У всех пользователей могут быть разные права доступа.

Это определяется *ролями* Kubernetes. Роль описывает определенный набор прав доступа. Kubernetes поставляется с некоторыми готовыми ролями, с которых можно начать. Например, роль `cluster-admin`, предназначенная для администраторов, имеет право на чтение и изменение любого ресурса в кластере. Для сравнения, роль `view` выводит списки объектов в заданном пространстве имен и просматривает их по отдельности, но не может их изменять.

Существует и такая опция — определение роли на уровне пространства имен (с помощью объекта `Role`) или для всего кластера (с использованием объекта `ClusterRole`). Вот пример манифеста `ClusterRole`, который выдает доступ к конфиденциальным данным в любом пространстве имен:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

Привязка ролей к пользователям

Как же соединить пользователя и роль? Это можно сделать с помощью *привязки ролей*. Как и в предыдущем примере, вы можете создавать объекты `RoleBinding` и `ClusterRoleBinding`, действующие на уровне пространства имен и, соответственно, всего кластера.

Ниже показан манифест `RoleBinding`, который назначает пользователю `daisy` роль `edit`, но лишь в пространстве имен `demo`:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: daisy-edit
  namespace: demo
subjects:
- kind: User
  name: daisy
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: edit
  apiGroup: rbac.authorization.k8s.io
```

В Kubernetes права доступа являются *добавочными*: вначале пользователи не имеют никаких прав и добавляют их с помощью объектов `Role` и `RoleBinding`. Если права уже выданы, их нельзя забрать.

Более подробно о RBAC, а также об имеющихся ролях и правах доступа можно почитать в документации Kubernetes (kubernetes.io/docs/reference/access-authn-authz/rbac).

Какие роли вам нужны

Какие же роли и привязки стоит использовать в своем кластере? Заранее определенные роли `cluster-admin`, `edit` и `view`, вероятно, будут соответствовать большинству требований. Чтобы узнать, какими правами доступа обладает та или иная роль, используйте команду `kubectl describe`:

```
kubectl describe clusterrole/edit
Name:         edit
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate=true
PolicyRule:
  Resources   ... Verbs
  ---------   ... -----
  bindings    ... [get list watch]
  configmaps  ... [create delete deletecollection get list patch update watch]
  endpoints   ... [create delete deletecollection get list patch update watch]
  ...
```

Роли можно создавать как для определенных людей или должностей внутри организации (например, роль «разработчик»), так и для отдельных команд (например, «обеспечение качества» или «безопасность»).

Обращение с ролью cluster-admin

Будьте осторожны с тем, кому выдавать доступ к роли `cluster-admin`: это администратор кластера, эквивалент пользователя `root` в системах Unix. Он может делать что угодно и с чем угодно. Никогда не выдавайте эту роль пользователям, которые не являются системными администраторами кластера, и особенно служебным учетным записям приложений, имеющим доступ к Интернету, например Kubernetes Dashboard (см. подраздел «Kubernetes Dashboard» на с. 275).

Не следует решать проблемы путем назначения роли cluster-admin без серьезных на то оснований. Несколько плохих советов по этому поводу вы найдете на таких сайтах, как Stack Overflow. Сталкиваясь с ошибками доступа, пользователи часто выдают своему приложению роль cluster-admin. Не делайте так. Действительно, это избавит вас от ошибок, однако все проверки безопасности будут игнорироваться, и ваш кластер может стать уязвимым для злоумышленников. Вместо этого выдайте своему приложению роль с минимальным набором привилегий, который позволит ему выполнять свою работу.

Приложения и развертывание

Приложениям, работающим в Kubernetes, обычно не требуется никаких прав доступа RBAC. По умолчанию все pod-оболочки выполняются от имени учетной записи `default`, принадлежащей их пространству имен и не связанной ни с какими ролями.

Если вашему приложению зачем-то нужен доступ к Kubernetes API (например, если речь об инструменте мониторинга, которому необходим список pod-оболочек), создайте для него отдельную служебную запись, свяжите ее с необходимой ролью (например, `view`) с помощью `RoleBinding` и ограничьте определенными пространствами имен.

Но что насчет прав доступа, которые нужны для развертывания приложения в кластере? Безопаснее всего полностью делегировать эту процедуру инструменту непрерывного развертывания (см. главу 14). У него может быть отдельная служебная учетная запись с правами на создание и удаление pod-оболочек в определенном пространстве имен.

Для этого идеально подходит роль `edit`. Пользователи с такой ролью могут создавать и удалять ресурсы в своем пространстве имен, однако им запрещено создавать новые роли и выдавать права доступа другим пользователям.

Если у вас нет автоматической системы развертывания и разработчики сами развертывают свой код в кластере, им требуются права на редактирование в заданном пространстве имен. Выдавайте права для каждого приложения по отдельности, а не для всего кластера целиком. У людей, которым не нужно развертывать приложения, по умолчанию должна быть только роль `view`.

Рекомендуемый подход

Убедитесь в том, что система RBAC включена во всех ваших кластерах. Выдавайте права cluster-admin только тем пользователям, которым действительно может понадобиться возможность уничтожить любое содержимое кластера. Если вашему приложению требуется доступ к ресурсам кластера, создайте для него служебную учетную запись и привяжите ее к роли, у которой есть только необходимые права доступа и только в нужном пространстве имен.

Решение проблем с RBAC

Если вы имеете дело со старым сторонним приложением, которое не знает о RBAC, или все еще в процессе определения прав для вашего собственного приложения, у вас могут возникнуть ошибки доступа RBAC. Как они выглядят?

Когда приложение выполняет API-запрос к тому, к чему у него нет доступа (например, вывод списка узлов), оно получает от API-сервера ответ с ошибкой Forbidden (HTTP-код 403):

```
Error from server (Forbidden): nodes.metrics.k8s.io is forbidden: User
"demo" cannot list nodes.metrics.k8s.io at the cluster scope.
```

Если приложение не записывает такую информацию или вы не знаете точно, кто виновен в этой ошибке, можете проверить журнал API-сервера (подробнее об этом в подразделе «Просмотр журнальных записей контейнера» на с. 167). Там можно найти подобные сообщения со строкой `RBAC DENY`, в которой описывается ошибка:

```
kubectl logs -n kube-system -l component=kube-apiserver | grep "RBAC DENY"
RBAC DENY: user "demo" cannot "list" resource "nodes" cluster-wide
```

(Вы не сможете сделать это в кластере GKE или любом другом управляемом сервисе Kubernetes, который не предоставляет доступа к управляющему уровню: чтобы узнать, как получить доступ к журнальным записям API-сервера, сверьтесь с документацией своего провайдера.)

Систему RBAC принято считать сложной, но на самом деле это не так. Просто выдавайте пользователям минимальные необходимые им привилегии, следите за ролью `cluster-admin` — и все будет хорошо.

Сканирование безопасности

Если вы запускаете в своем кластере стороннее программное обеспечение, стоит проверить его на проблемы с безопасностью или вредоносный код. Но и в ваших собственных контейнерах может быть ПО, о котором вы не знаете, — это тоже требует проверки.

Clair

Clair (https://github.com/coreos/clair) — это открытый сканер контейнеров, разработанный в рамках проекта CoreOS. Он статически анализирует образы контейнеров перед их запуском, чтобы убедиться в том, что они не содержат никакого ПО или своих версий с известными уязвимостями.

Clair можно запускать вручную для поиска проблем в отдельных контейнерах или интегрировать в ваш процесс непрерывной доставки для проверки всех образов перед их развертыванием (см. главу 14).

Как вариант, Clair может подключиться к вашему реестру контейнеров и сканировать все образы, которые в него загружаются, сообщая о проблемах.

Стоит упомянуть, что базовым образам, таким как `alpine`, не следует доверять безоговорочно. Clair поставляется с множеством проверок для многих популярных базовых образов, поэтому сразу же даст знать, если вы используете один из образов с известными уязвимостями.

Aqua

Aqua Security Platform от компании Aqua (www.aquasec.com/products/aqua-cloud-native-security-platform) — это коммерческое решение с полным набором услуг. Оно позволяет организациям сканировать контейнеры на уязвимости, вредоносный код и подозрительную активность, а также следить за соблюдением политик и соответствием нормативным требованиям.

Как и следовало ожидать, платформа Aqua интегрируется с вашим реестром контейнеров, процессами CI/CD и различными системами оркестрации, включая Kubernetes.

Кроме того, Aqua предлагает бесплатный инструмент под названием MicroScanner (github.com/aquasecurity/microscanner), который можно добавлять в образы контейнеров для сканирования установленных пакетов на предмет уязвимостей с использованием той же базы данных, что и Aqua Security Platform.

MicroScanner устанавливается путем добавления в Dockerfile. Например:

```
ADD https://get.aquasec.com/microscanner /
RUN chmod +x /microscanner
RUN /microscanner <TOKEN> [--continue-on-failure]
```

MicroScanner выводит список обнаруженных уязвимостей в формате JSON. Список можно прочитать и использовать для отправки уведомлений с помощью других инструментов.

Еще один полезный инструмент с открытым исходным кодом от Aqua `kube-hunter` (kube-hunter.aquasec.com) создан для поиска проблем с безопасностью в самом кластере Kubernetes. Если запустить инструмент в виде контейнера на компьютере за пределами вашего кластера (как это сделал бы злоумышленник), он проведет проверку на наличие разного рода проблем, например раскрытых адресов электронной почты в сертификатах, неиспользуемых приборных панелей, открытых портов и конечных точек и т. д.

Anchore Engine

Anchore Engine (github.com/anchore/anchore-engine) — это инструмент с открытым исходным кодом не только для сканирования образов контейнеров на известные уязвимости, но и для идентификации всего, что присутствует в контейнере, включая библиотеки, конфигурационные файлы и права доступа. С его помощью контейнеры можно проверять на соответствие пользовательским политикам: например, вы можете заблокировать любые образы, которые содержат конфиденциальные учетные данные или исходный код приложения.

Рекомендуемый подход

Не запускайте контейнеры, которые взяты из сомнительных источников или имеют неизвестное содержимое. Применяйте инструменты сканирования наподобие Clair или MicroScanner ко всем контейнерам — даже тем, которые вы собрали сами. Это поможет убедиться в том, что ни в каких базовых образах или зависимостях нет известных уязвимостей.

Резервное копирование

Вам, наверное, интересно, имеет ли место резервное копирование в облачно-ориентированных архитектурах. В конце концов, платформа Kubernetes по своей природе является надежной и способна выдержать потерю сразу нескольких узлов без потери состояния или даже заметного снижения производительности.

К тому же Kubernetes — это декларативная система типа «инфраструктура как код». Все ее ресурсы описываются в виде данных, хранящихся в надежной базе данных *(etcd)*. Если какие-то pod-оболочки будут случайно удалены, развертывание, которое за них отвечает, воссоздаст их из спецификации, размещенной в базе данных.

Нужно ли выполнять резервное копирование в Kubernetes

Так есть ли по-прежнему необходимость в резервном копировании? Есть. Например, данные, хранящиеся в постоянных томах, подвержены сбоям (см. подраздел «Постоянные тома» на с. 201). И хотя ваш облачный провайдер может предоставлять высокодоступные тома (например, с репликацией данных между разными зонами доступности), это не то же самое, что резервная копия.

Еще раз повторим эту мысль, потому что она неочевидная.

Репликация — это не резервное копирование. Она может уберечь вас от сбоя в томе внутреннего хранилища, но не защитит от случайного удаления тома, если вы нажмете не ту кнопку в веб-консоли.

Репликация также не поможет в ситуации, когда неправильно настроенное приложение перезаписывает свои данные или когда системный администратор запускает команду не с теми переменными среды и случайно удаляет промышленную базу данных вместо той, что используется в разработке (такое действительно случается (thenewstack.io/junior-dev-deleted-production-database), и, вероятно, чаще, чем кто-либо готов признать).

Резервное копирование etcd

Как мы уже видели в подразделе «Высокая доступность» на с. 67, Kubernetes хранит все свое состояние в базе данных *etcd*, поэтому любые ее сбои или потеря информации могут оказаться катастрофическими. Уже одной этой причины достаточно, чтобы рекомендовать к использованию управляемые сервисы, которые гарантируют высокую доступность *etcd* и управляющего уровня в целом (см. подраздел «По возможности используйте Kubernetes в виде управляемого сервиса» на с. 82).

Если вы сами запускаете свои ведущие узлы, вся ответственность за кластеризацию, репликацию и резервное копирование *etcd* ложится на вас. Даже при обычном снимке данных определенное время занимают его извлечение и проверка, а также перестройка кластера и восстановление информации. И пока вы это делаете, ваш кластер, скорее всего, будет недоступен или испытает серьезную потерю производительности.

Рекомендуемый подход

Чтобы иметь кластеризацию и резервное копирование для etcd на своих ведущих узлах, используйте управляемые сервисы или решения под ключ. Если этим вы занимаетесь сами, убедитесь в том, что достаточно компетентны для подобных действий. Обеспечение устойчивости etcd — работа для специалиста, ведь если что-то пойдет не так, последствия могут быть серьезными.

Резервное копирование состояния ресурсов

Помимо сбоев, в *etcd* также актуальна тема сохранения состояния отдельных ресурсов. Например, как воссоздать развертывание, которое было удалено случайно?

В этой книге мы постоянно подчеркиваем ценность парадигмы *«инфраструктура как код»* и советуем всегда управлять ресурсами Kubernetes декларативно, применяя YAML-манифесты или чарты HELM в системе контроля версий.

Таким образом, в теории, чтобы воссоздать состояние всех рабочих заданий в кластере, вам нужно будет взять подходящие репозитории и применить все ресурсы, которые в них находятся. *Но это в теории.*

Резервное копирование состояния кластера

На практике не все, что находится в системе контроля версий, выполняется в вашем кластере в тот или иной момент. Какие-то приложения могут быть выведены из эксплуатации или заменены новыми версиями, а другие могут быть не готовы к развертыванию.

На страницах этой книги мы советуем избегать редактирования ресурсов напрямую: лучше применять изменения из обновленных файлов манифеста (см. подраздел «Когда не следует использовать императивные команды» на с. 164). Но люди не всегда следуют хорошим советам (на что постоянно сетуют консультанты).

Как бы то ни было, во время начального развертывания и тестирования приложений инженеры вполне могут подправить на ходу такие параметры, как количество реплик или принадлежность к узлам, а в системе контроля версий сохранить уже проверенные значения.

Представьте, что ваш кластер полностью выключен или что все его ресурсы удалены (надеемся, что это маловероятная ситуация, которая тем не менее полезна в качестве мысленного эксперимента). Как бы вы его воссоздали?

Даже если у вас есть превосходно спроектированная и актуальная система автоматизации, способная заново развернуть все в свежем кластере, — откуда вы *знаете*, что состояние этого нового кластера совпадет с тем, который вы потеряли?

Для уверенности в этом можно сделать снимок активного кластера, к которому можно будет обратиться в случае возникновения проблем.

Крупные и мелкие сбои

Потеря целого кластера Kubernetes — маловероятный сценарий: тысячи людей тяжело трудятся над тем, чтобы подобное не случилось.

Более вероятной является ситуация, когда вы (или новичок в вашей команде) случайно удалите пространство имен, выключите развертывание или, указав в команде `kubectl delete` неправильный набор меток, удалите больше, чем планировалось.

Какой бы ни была причина, катастрофы случаются, поэтому далее мы рассмотрим инструмент для резервного копирования, который поможет вам их избежать.

Velero

Velero — это бесплатный и открытый инструмент (ранее известный как Ark), умеющий сохранять и восстанавливать состояние вашего кластера и постоянные данные.

Velero работает в кластере и подключается к сервису облачного хранилища по вашему выбору (например, Amazon S3 или Azure Storage).

Чтобы настроить Velero на вашей платформе, следуйте инструкциям (velero.io).

Конфигурация Velero

Прежде чем использовать Velero, вам необходимо создать в своем кластере Kubernetes объект `BackupStorageLocation` и указать, где нужно хранить резервные копии (например, в бакете облачного хранилища AWS S3). Вот пример конфигурации Velero с резервным копированием в бакет `demo-backup`:

```
apiVersion: velero.io/v1
kind: BackupStorageLocation
metadata:
  name: default
  namespace: velero
spec:
  provider: aws
  objectStorage:
    bucket: demo-backup
  config:
    region: us-east-1
```

У вас должно быть как минимум одно место хранения с именем `default`, а в дополнение к нему можно создавать другие, с любыми именами на ваш выбор.

Velero может также сохранять содержимое ваших постоянных томов. Чтобы определить место их хранения, создайте объект `VolumeSnapshotLocation`:

```
apiVersion: velero.io/v1
kind: VolumeSnapshotLocation
metadata:
  name: aws-default
  namespace: velero
spec:
  provider: aws
  config:
    region: us-east-1
```

Создание резервной копии с помощью Velero

Когда вы создаете резервную копию с помощью команды `velero backup`, сервер Velero обращается к Kubernetes API, чтобы извлечь все ресурсы, соответствующие заданному селектору (по умолчанию копируются все ресурсы). Вы можете сохранить набор пространств имен или целый кластер:

```
velero backup create demo-backup --include-namespaces demo
```

Все эти ресурсы экспортируются в заданный файл в бакете вашего облачного хранилища в соответствии с предоставленной вами конфигурацией `BackupStorageLocation`. Метаданные и содержимое постоянных томов копируются в место, описанное в `VolumeSnapshotLocation`.

Как вариант, вы можете создать резервную копию всего кластера *за исключением* заданных пространств имен (таких как `kube-system`) и запланировать автоматическое резервное копирование, чтобы Velero сохранял состояние вашего кластера, например, каждую ночь или ежечасно.

Все резервные копии Velero являются самодостаточными, а не инкрементальными. Поэтому для восстановления вам нужен лишь файл с самой свежей версией.

Восстановление данных

Список доступных резервных копий выводится с помощью команды `velero backup get`:

```
velero backup get
NAME         STATUS      CREATED                         EXPIRES   SELECTOR
demo-backup  Completed   2018-07-14 10:54:20 +0100 BST   29d       <none>
```

Чтобы увидеть содержимое отдельной резервной копии, введите `velero backup download`:

```
velero backup download demo-backup
Backup demo-backup has been successfully downloaded to
$PWD/demo-backup-data.tar.gz
```

Загруженный файл представляет собой архив `tar.gz`, который распаковывается и просматривается с помощью стандартных инструментов. Например, если вам нужен только манифест определенного ресурса, извлеките его из файла резервной копии и восстановите отдельно с помощью команды `kubectl apply -f`.

Чтобы инициировать восстановление всей резервной копии, обратитесь к команде `velero restore`: Velero воссоздаст все ресурсы и тома, описанные в заданном снимке, и пропустит те из них, которые уже существуют.

Если ресурс *существует*, но отличается от того, что находится в резервной копии, Velero вас об этом предупредит, но перезаписывать оригинал не станет. Имея потребность сбросить состояние активного развертывания к тому, которое было на момент снятия последнего снимка, сначала удалите развертывание, а только затем восстановите.

Если вы восстанавливаете состояние пространства имен, удалите это пространство и затем восстановите резервную копию.

Процедуры и проверки восстановления

Вы должны подробно и пошагово задокументировать процедуру восстановления данных из резервной копии и убедиться, что все ваши коллеги знают, где можно найти этот документ. Катастрофы обычно случаются в неудобное время, когда ключевые люди недоступны и все вокруг паникуют, поэтому ваша процедура должна быть настолько четкой и точной, чтобы ее мог выполнить человек, незнакомый с Velero или даже с Kubernetes.

Проводите ежемесячные учения, поручая разным членам команды выполнять процедуру восстановления на примере временного кластера. Это подтвердит надежность ваших рабочих копий, корректность вашей процедуры восстановления и тот факт, что все знают, как это делается.

Планирование резервного копирования в Velero

Любое резервное копирование должно быть автоматизировано, и Velero не является исключением. Запланировать регулярное создание резервных копий можно с помощью команды `velero schedule create`:

```
velero schedule create demo-schedule --schedule="0 1 * * *" --include-namespaces
demo
Schedule "demo-schedule" created successfully.
```

Аргумент `schedule` определяет время выполнения процедуры в формате Unix `cron` (см. подраздел «Задания Cronjob» на с. 222). Например, если указать `0 1 * * *`, резервное копирование будет производиться каждую ночь в 01:00.

Чтобы просмотреть запланированные процедуры, используйте команду `velero schedule get`:

```
velero schedule get
NAME          STATUS  CREATED     SCHEDULE    BACKUP TTL LAST BACKUP SELECTOR
demo-schedule Enabled 2018-07-14  * 10 * * *  720h0m0s   10h ago     <none>
```

В поле `BACKUP TTL` указано, как долго будет храниться резервная копия (до удаления). Значение по умолчанию — 720 часов, что эквивалентно одному месяцу.

Другие способы применения Velero

Инструмент Velero чрезвычайно полезен для восстановления после катастрофы, но с его помощью также можно перемещать ресурсы и данные из одного кластера в другой. Этот процесс иногда называют *lift and shift* («поднять и передвинуть»).

Регулярное резервное копирование с применением Velero помогает понять, как меняется потребление ресурсов Kubernetes: например, вы можете сравнить текущее состояние с тем, которое было месяц, полгода или год назад.

Снимки могут быть и полезным источником информации для аудита, когда вам надо посмотреть, что выполнялось в вашем кластере в конкретное время и как/когда состояние кластера изменилось.

Рекомендуемый подход

Используйте Velero для регулярного резервного копирования состояния и постоянных данных вашего кластера (как минимум каждую ночь). Проверяйте восстановление хотя бы раз в месяц.

Мониторинг состояния кластера

Мониторинг облачно-ориентированных приложений — это обширная тема, которая, как вы увидите в главе 15, включает в себя такие действия, как наблюдаемость, сбор показателей, ведение журнала, трассирование и традиционный мониторинг методом черного ящика.

Но в данной главе мы сосредоточимся только на мониторинге самого кластера Kubernetes: его работоспособности, состояния отдельных узлов, загруженности оборудования и хода выполнения его рабочих заданий.

kubectl

С бесценной командой `kubectl` мы познакомили вас еще в главе 2, но рассказали не обо всех ее возможностях. Помимо того что это универсальный инструмент для администрирования ресурсов Kubernetes, `kubectl` может и предоставлять полезную информацию о состоянии компонентов кластера.

Состояние управляющего уровня

Команда `kubectl get componentstatuses` (сокращенно `kubectl get cs`) выводит сведения о работоспособности компонентов управляющего уровня — планировщика, диспетчера контроллеров и *etcd*:

```
kubectl get componentstatuses
NAME                 STATUS    MESSAGE           ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-0               Healthy   {"health": "true"}
```

Серьезные проблемы с любым из компонентов управляющего уровня обычно быстро дают о себе знать, однако все равно очень удобно иметь возможность все проверить и получить некий обобщенный показатель работоспособности кластера.

Если какой-либо компонент управляющего уровня имеет состояние, отличное от `Healthy`, его нужно исправить. С управляемыми сервисами Kubernetes такой проблемы возникнуть не должно, но если вы размещаете свой кластер самостоятельно, вам придется самим об этом позаботиться.

Состояние узла

Еще одна полезная команда — `kubectl get nodes` — выводит список всех узлов в вашем кластере, включая их состояние и версию Kubernetes:

```
kubectl get nodes
NAME                 STATUS   ROLES    AGE   VERSION
docker-for-desktop   Ready    master   5d    v1.10.0
```

Поскольку кластеры Docker Desktop содержат лишь один узел, данный вывод не очень информативен. В качестве более реалистичного примера возьмем небольшой кластер Google Kubernetes Engine:

```
kubectl get nodes
NAME                                          STATUS   ROLES    AGE   VERSION
gke-k8s-cluster-1-n1-standard-2-pool--8l6n    Ready    <none>   9d    v1.10.2-gke.1
gke-k8s-cluster-1-n1-standard-2-pool--dwtv    Ready    <none>   19d   v1.10.2-gke.1
gke-k8s-cluster-1-n1-standard-2-pool--67ch    Ready    <none>   20d   v1.10.2-gke.1
...
```

Обратите внимание на то, что в выводе команды `get nodes` для Docker Desktop узел имел *роль* `master`. Естественно, единственный узел должен быть ведущим (и одновременно единственным рабочим).

В Google Kubernetes Engine и некоторых других управляемых сервисах Kubernetes у вас нет прямого доступа к ведущим узлам. Соответственно, команда `kubectl get nodes` выводит лишь рабочие узлы (на то, что они рабочие, указывает роль `<none>`).

Состояние `NotReady` какого-либо из узлов сообщает о наличии проблемы. В подобном случае следует перезагрузить узел, а если это не поможет, произвести дальнейшую отладку (или же просто удалить данный узел и создать вместо него новый).

Для подробного исследования проблем с узлами используйте команду `kubectl describe node`, которая выводит дополнительную информацию:

```
kubectl describe nodes/gke-k8s-cluster-1-n1-standard-2-pool--8l6n
```

В этом случае будут выведены сведения о вычислительной емкости узла и ресурсах, занятых pod-оболочками в настоящее время.

Рабочие задания

Как вы можете помнить из подраздела «Обращение к кластеру с помощью kubectl» на с. 100, `kubectl` также позволяет выводить список всех pod-оболочек (или любых других ресурсов) в вашем кластере. В том примере мы вывели только оболочки из пространства имен по умолчанию. Чтобы сделать то же самое, но в масштабах всего кластера, используйте флаг `--all-namespaces`:

```
kubectl get pods --all-namespaces
NAMESPACE      NAME                             READY   STATUS             RESTARTS   AGE
cert-manager   cert-manager-cert-manager-55     1/1     Running            1          10d
pa-test        permissions-auditor-15281892     0/1     CrashLoopBackOff   1720       6d
freshtracks    freshtracks-agent-779758f445     3/3     Running            5          20d
...
```

Это даст вам хороший обзор того, что запущено в кластере, и вообще любых проблем, связанных с `Pod`-объектами. Если какая-либо pod-оболочка не находится в состоянии `Running` (как `permissions-auditor` в нашем примере), придется разбираться дальше.

В столбце `READY` показано, сколько контейнеров в pod-оболочке на самом деле работает и какое число было задано в конфигурации. Например, объект `freshtracks-agent` имеет значение `3/3`: запущено три контейнера из трех, поэтому все хорошо.

С другой стороны, `permissions-auditor` показывает `0/1`: был задан один контейнер, но выполняется ноль. Причина приводится в столбце `STATUS`: `CrashLoopBackOff`. Контейнеру не удается корректно запуститься.

Если контейнер откажет, Kubernetes периодически будет пытаться его перезапустить: начнет через 10 секунд, с каждой попыткой увеличивая интервал вдвое, пока тот не достигнет пяти минут. Это называется *экспоненциальной выдержкой*, отсюда и состояние `CrashLoopBackOff`.

Загруженность процессора и памяти

Еще один способ представления кластера дает команда `kubectl top`. Она показывает емкость процессора и объем памяти на каждом из узлов и какая доля этих ресурсов сейчас используется:

```
kubectl top nodes
NAME                          CPU(cores)  CPU%  MEMORY(bytes)  MEMORY%
gke-k8s-cluster-1-n1-...8l6n  151m        7%    2783Mi         49%
gke-k8s-cluster-1-n1-...dwtv  155m        8%    3449Mi         61%
gke-k8s-cluster-1-n1-...67ch  580m        30%   3172Mi         56%
...
```

Она также может показать, сколько ресурсов процессора и памяти занимает каждая pod-оболочка:

```
kubectl top pods -n kube-system
NAME                                  CPU(cores)  MEMORY(bytes)
event-exporter-v0.1.9-85bb4fd64d-2zjng  0m          27Mi
fluentd-gcp-scaler-7c5db745fc-h7ntr     10m         27Mi
fluentd-gcp-v3.0.0-5m627                11m         171Mi
...
```

Консоль облачного провайдера

Если вы пользуетесь сервисом по управлению Kubernetes, предоставляемым облачным провайдером, у вас должен быть доступ к веб-консоли с полезной информацией о кластере, его узлах и рабочих заданиях.

Например, консоль Google Kubernetes Engine (GKE) перечисляет все ваши кластеры и о каждом из них — о пулах узлов и т. д. — дает детальные сведения (рис. 11.1).

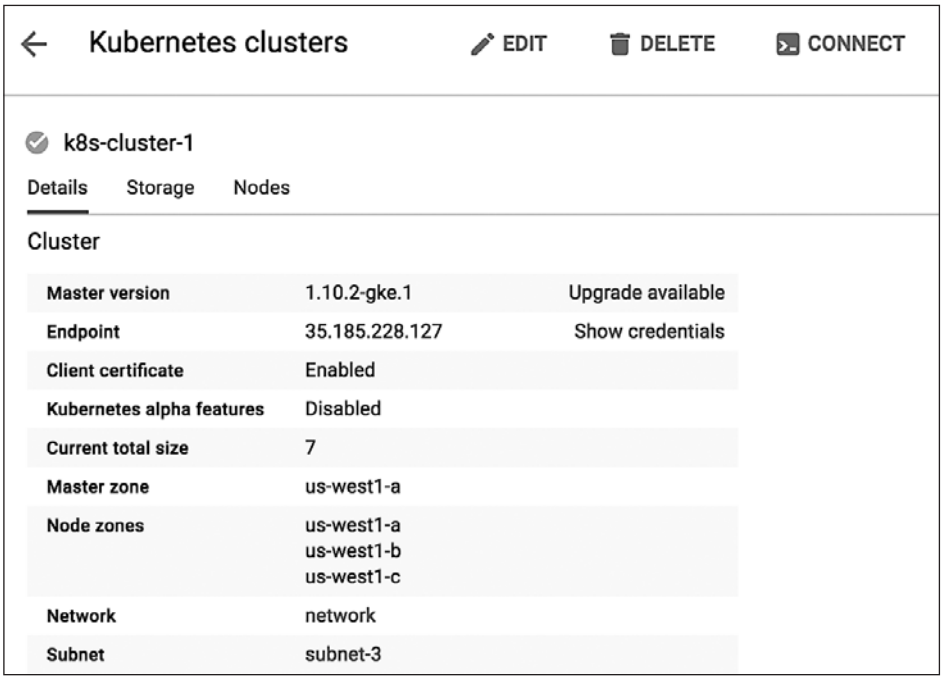

Рис. 11.1. Консоль Google Kubernetes Engine

Вы также можете выводить списки рабочих заданий, сервисов и конфигурационных параметров кластера. В основном это та же информация, которую можно получить с помощью `kubectl`, но при этом консоль GKE позволяет выполнять и административные задачи: создавать кластеры, обновлять узлы и многое другое, что может потребоваться в ходе ежедневной работы с Kubernetes.

Azure Kubernetes Service, AWS Elastic Container Service for Kubernetes и другие провайдеры управляемых сервисов предоставляют похожие средства. Мы рекомендуем ознакомиться с консолью управления вашего провайдера, так как вы будете очень часто ее использовать.

Kubernetes Dashboard

Kubernetes Dashboard (kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard) — это пользовательский веб-интерфейс для кластеров Kubernetes (рис. 11.2). Если вы сами занимаетесь размещением своего кластера и не используете управляемые сервисы, Kubernetes Dashboard позволит получить примерно ту же информацию, что и облачные консоли.

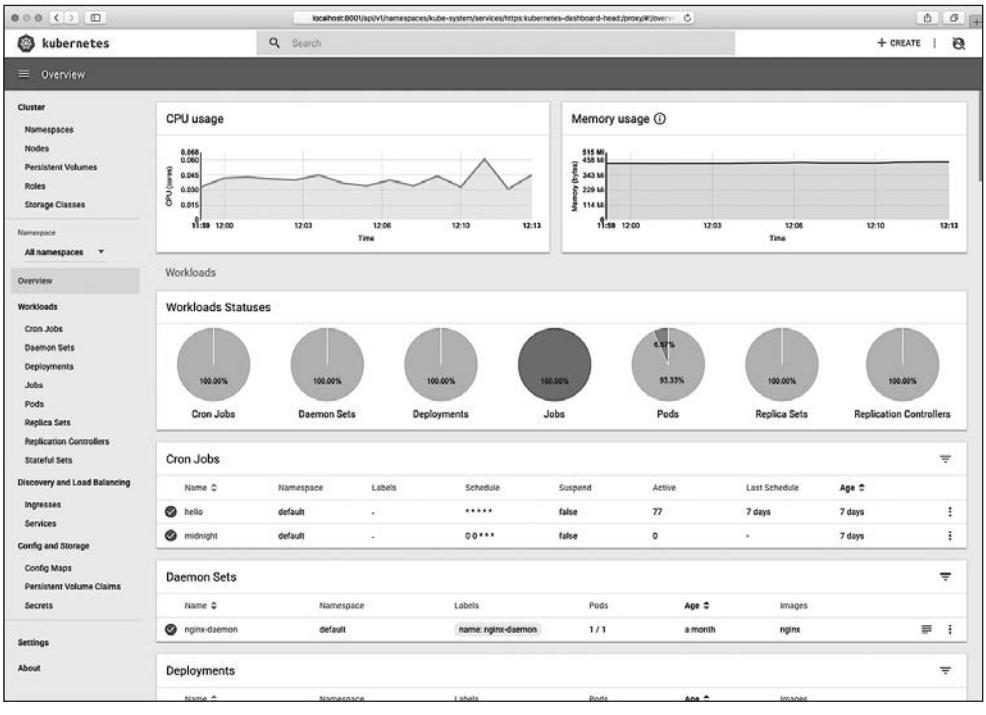

Рис. 11.2. Kubernetes Dashboard выводит полезные сведения о вашем кластере

Как и следовало ожидать, Dashboard позволяет просматривать состояние ваших кластеров, узлов и рабочих заданий практически так же, как и `kubectl`, но только в графическом интерфейсе. А еще с помощью этой консоли можно создавать и удалять ресурсы.

Поскольку консоль Dashboard выводит обширный набор информации о кластере и рабочих заданиях, очень важно обеспечить ей надежную защиту и никогда не открывать к ней публичный доступ из Интернета. Консоль позволяет просматривать содержимое ресурсов `ConfigMap` и `Secret`, в которых могут храниться учетные данные и криптографические ключи, поэтому контролировать доступ к панели мониторинга нужно так же тщательно, как и к самим конфиденциальным данным.

В 2018 году фирма RedLock, занимающаяся безопасностью, обнаружила сотни консолей Kubernetes Dashboard (redlock.io/blog/cryptojacking-tesla), в том числе и принадлежащие Tesla, Inc., которые были доступны из Интернета и не защищались никаким паролем. Таким образом фирма RedLock смогла извлечь конфиденциальные учетные данные для входа в облако и получить еще больше конфиденциальной информации.

Рекомендуемый подход

Не используйте консоль Kubernetes Dashboard без особой надобности (например, если у вас уже есть консоль Kubernetes, предоставляемая управляемым сервисом, таким как GKE). Если же вы с ней работаете, убедитесь, что она имеет минимальные привилегии (blog.heptio.com/on-securing-the-kubernetes-dashboard-16b09b1b7aca) и к ней нет доступа из Интернета. Обращайтесь к консоли только через kubectl proxy.

Weave Scope

Weave Scope — это отличный инструмент для визуализации и мониторинга, который показывает вам узлы, контейнеры и процессы в режиме реального времени. Вы также можете просматривать показатели и метаданные или даже запускать или останавливать контейнеры.

kube-ops-view

В отличие от Kubernetes Dashboard kube-ops-view (github.com/hjacobs/kube-ops-view) не пытается управлять кластером общего назначения. Этот инструмент просто визуализирует то, что происходит в вашем кластере, и показывает, какие в нем есть узлы, сколько ресурсов процессора и памяти занято, сколько pod-оболочек и с каким состоянием выполняется на каждом из узлов (рис. 11.3).

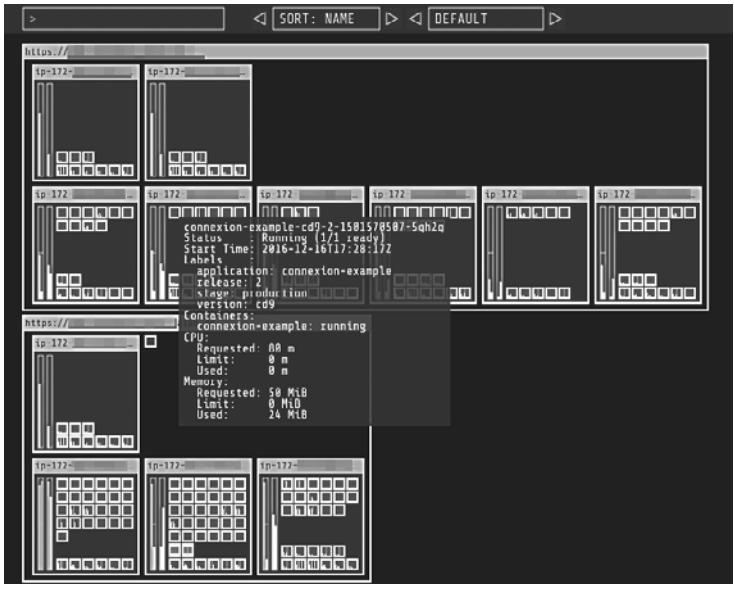

Рис. 11.3. kube-ops-view дает оперативное представление о вашем кластере Kubernetes

node-problem-detector

node-problem-detector (github.com/kubernetes/node-problem-detector) — дополнение для Kubernetes, способное обнаруживать проблемы на уровне узлов и сообщать о них. Речь идет об аппаратных сбоях, таких как ошибки работы процессора или памяти, повреждение файловой системы и зависание среды выполнения контейнера.

На сегодняшний день node-problem-detector сообщает о проблемах путем отправки событий в Kubernetes API и поставляется вместе с клиентской библиотекой на Go, которую можно использовать для интеграции собственных инструментов.

Пока Kubernetes не позволяет каким-либо образом реагировать на события, полученные из node-problem-detector, но в будущем интеграция может улучшиться и планировщик, например, сможет избегать развертывания pod-оболочек на проблемных узлах.

Использование node-problem-detector — отличный способ получить общий обзор своего кластера и того, что он делает. Это не замена Dashboard или специализированных инструментов для мониторинга, а хорошее дополнение к ним.

Дополнительный материал

Безопасность Kubernetes — сложная и узкоспециализированная тема, и здесь мы ее едва затронули. Она заслуживает отдельной книги... Которая теперь существует! Замечательное издание *Kubernetes Security* (www.oreilly.com/library/view/kubernetes-security/9781492039075) (O'Reilly) авторства Лиз Райс и Майкла Хаузенбласа охватывает безопасную настройку кластера, безопасность контейнеров, управление конфиденциальными данными и т. д. Настоятельно рекомендуется к прочтению.

Резюме

Обеспечение безопасности — это не продукт или конечная цель, а непрерывный процесс, который требует знаний, вдумчивости и внимания. Безопасность контейнеров в этом смысле ничем не отличается, и механизмы для ее достижения находятся в вашем распоряжении. Материала, усвоенного из этой главы, должно быть достаточно для безопасной конфигурации контейнеров в Kubernetes. Но, надеемся, вы понимаете, что это лишь начало, а не конец вашего процесса обеспечения безопасности.

Основные моменты, о которых стоит помнить.

- Система RBAC предоставляет гибкий механизм управления правами доступа в Kubernetes. Убедитесь в том, что она включена, и используйте роли RBAC для выдачи определенным пользователям и приложениям тех минимальных привилегий, которые нужны им для работы.
- Контейнеры не защищены волшебным образом от вредоносных кодов и других проблем с безопасностью. Используйте инструмент для сканирования, чтобы проверять контейнеры, работающие в промышленных условиях.
- Kubernetes — платформа прекрасная и так далее, но резервное копирование никто не отменял. Используйте Velero для сохранения данных и состояния вашего кластера. Этот инструмент также подходит для перемещения объектов из одного кластера в другой.
- `kubectl` — мощный инструмент для исследования и оповещения обо всех аспектах деятельности вашего кластера и его рабочих заданий. Познакомьтесь с `kubectl` поближе — вместе вы будете проводить много времени.
- Используйте веб-консоль вашего провайдера Kubernetes и kube-ops-view для графического представления происходящего в кластере. Если вы работаете с консолью Kubernetes Dashboard, позаботьтесь о ее защите так же тщательно, как делаете это с облачными учетными данными и криптографическими ключами.

12 Развертывание приложений Kubernetes

> Я лежу на спине, удивленный, насколько спокойным и сосредоточенным себя чувствую, пристегнутый к двум с лишним тоннам взрывчатых веществ.
>
> *Рон Гаран, астронавт*

В этой главе мы поговорим о том, как превратить файлы с манифестами в рабочие приложения. Вы научитесь собирать чарты Helm для своих проектов и узнаете о некоторых альтернативных инструментах для управления манифестами: ksonnet, kustomize, kapitan и kompose.

Построение манифестов с помощью Helm

В главе 2 было показано, как развертывать и администрировать приложения с помощью ресурсов Kubernetes, созданных из манифестов в формате YAML. Ничто не мешает вам управлять приложениями исключительно посредством необработанных (сырых) файлов YAML, однако это не лучший способ. Помимо сложности обращения со всеми этими файлами, встает еще и вопрос дистрибуции.

Предположим, вы хотите, чтобы другие люди могли запускать ваше приложение в своих собственных кластерах. Вы можете передать им манифесты файлов, однако новым пользователям неизбежно придется подгонять некоторые параметры под свою среду.

Для этого потребуется создать копию конфигурации Kubernetes, найти, где определяются различные настройки (которые могут дублироваться в разных местах), и отредактировать их.

Со временем пользователи должны будут сохранять уже свои копии файлов. Но когда вы выпустите обновление, его придется загружать и вручную объединять с локальными изменениями.

Рано или поздно такие действия станут утомительными. Поэтому нужна возможность отделять необработанные файлы манифестов от определенных параметров и переменных, которые могут потребовать модификации с вашей стороны или со стороны любых других пользователей. В идеале все это можно было бы сделать доступным в стандартном формате, чтобы кто угодно мог загрузить и установить ваше приложение в кластере Kubernetes.

Тогда каждое приложение сможет предоставлять не только конфигурационные значения, но и свои зависимости от других программ и сервисов, а умный инструмент для управления пакетами установит и запустит приложения вместе со всеми их зависимостями, используя лишь одну команду.

В разделе «Helm: диспетчер пакетов для Kubernetes» на с. 102 мы познакомили вас с инструментом Helm и рассказали, как с его помощью можно устанавливать публичные чарты. Теперь взглянем на чарты Helm немного поближе и посмотрим, как они создаются.

Что внутри у чарта Helm

Откройте папку `hello-helm/k8s` в репозитории `demo`: в ней вы увидите, что хранится внутри чарта Helm.

Каждый чарт имеет стандартную структуру. Прежде всего, он находится внутри каталога с таким же именем (в данном случае `demo`):

```
demo
├── Chart.yaml
├── production-values.yaml
├── staging-values.yaml
├── templates
│   ├── deployment.yaml
│   └── service.yaml
└── values.yaml
```

Файл Chart.yaml

Кроме того, он содержит файл `Chart.yaml`, который определяет имя и версию чарта:

```
name: demo
sources:
  - https://github.com/cloudnativedevops/demo
version: 1.0.1
```

В `Chart.yaml` можно указать множество дополнительных полей — в том числе и ссылку на исходный код проекта, как здесь показано, — но обязательной информацией являются лишь имя и версия.

Файл values.yaml

Есть также файл под названием `values.yaml`, содержащий параметры, которые автор чарта позволяет менять пользователям:

```
environment: development
container:
  name: demo
  port: 8888
  image: cloudnatived/demo
  tag: hello
replicas: 1
```

Это чем-то напоминает файл YAML манифеста Kubernetes, но с одним важным отличием: файл `values.yaml` ограничен лишь форматом YAML и не имеет никакой предопределенной структуры. То, какие переменные устанавливать, какие имена и значения им присваивать, решаете вы сами.

В чарте может вообще не быть никаких переменных, но если таковые имеются, поместите их в `values.yaml` и затем ссылайтесь на них с других участков вашего чарта.

Пока не обращайте внимания на файлы `production-values.yaml` и `staging-values.yaml`: их назначение мы объясним чуть позже.

Шаблоны Helm

Так где же упоминаются эти переменные? Если заглянуть в папку `templates`, можно увидеть несколько знакомых файлов:

```
ls k8s/demo/templates
deployment.yaml service.yaml
```

Это такие же файлы манифестов для развертывания и сервиса, как и в предыдущем примере, только теперь они являются *шаблонами*. Вместо того чтобы напрямую что-то упоминать — например, имя контейнера, — шаблоны содержат заглушки, которые Helm заменит настоящими значениями из `values.yaml`.

Вот как выглядит шаблонное развертывание:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: {{ .Values.container.name }}-{{ .Values.environment }}
spec:
  replicas: {{ .Values.replicas }}
  selector:
    matchLabels:
      app: {{ .Values.container.name }}
  template:
    metadata:
      labels:
        app: {{ .Values.container.name }}
        environment: {{ .Values.environment }}
    spec:
      containers:
        - name: {{ .Values.container.name }}
          image: {{ .Values.container.image }}:{{ .Values.container.tag }}
          ports:
            - containerPort: {{ .Values.container.port }}
          env:
            - name: ENVIRONMENT
              value: {{ .Values.environment }}
```

Фигурные скобки указывают на место, куда диспетчер Helm должен подставить значение переменной. На самом деле это часть синтаксиса шаблонов Go.

Да, от Go никуда не деться. Kubernetes и Helm сами написаны на Go, поэтому неудивительно, что в чартах Helm применяется синтаксис этого языка.

Интерполяция переменных

В данном шаблоне упоминается несколько переменных:

```
...
metadata:
  name: {{ .Values.container.name }}-{{ .Values.environment }}
```

Весь фрагмент, включая фигурные скобки, будет *интерполирован* (то есть заменен) значениями `container.name` и `environment`, взятыми из `values.yaml`. Сгенерированный результат будет выглядеть так:

```
...
metadata:
  name: demo-development
```

Это очень удобно, потому что такие значения, как `container.name`, упоминаются в шаблоне по нескольку раз. Естественно, они присутствуют и в сервисе:

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.container.name }}-service-{{ .Values.environment }}
  labels:
    app: {{ .Values.container.name }}
spec:
  ports:
  - port: {{ .Values.container.port }}
    protocol: TCP
    targetPort: {{ .Values.container.port }}
  selector:
    app: {{ .Values.container.name }}
  type: ClusterIP
```

Вы видите, сколько раз упоминается, скажем, `.Values.container.name`: даже в простом чарте вроде этого вам необходимо многократно повторять одни и те же фрагменты информации. Благодаря переменным Helm подобного дублирования можно избежать. Например, чтобы поменять имя контейнера, достаточно отредактировать `values.yaml` и заново установить свой чарт: в результате изменение распространится на все шаблоны.

Шаблоны в Go имеют очень мощный синтаксис и далеко не ограничиваются простой подстановкой переменных, как в нашем примере: они поддерживают циклы, выражения, условия и даже вызов функций. Чарты Helm могут использовать эти возможности для создания довольно сложной конфигурации на основе вводных значений.

Больше о написании шаблонов можно почитать в документации Helm (helm.sh/docs/chart_template_guide).

Цитирование значений в шаблонах

Helm позволяет использовать функцию `quote` для автоматического цитирования значений в шаблонах:

```
name: {{.Values.MyName | quote }}
```

Цитировать следует только строковые значения. Не применяйте функцию `quote` для чисел, таких как номера портов.

Задание зависимостей

Что, если ваш чарт зависит от других чартов? Например, если приложение использует Redis, в его чарте Helm, возможно, придется указать чарт `redis` в качестве зависимости.

Это можно сделать в файле `requirements.yaml`:

```
dependencies:
  - name: redis
    version: 1.2.3
  - name: nginx
    version: 3.2.1
```

Теперь выполните команду `helm dependency update`: Helm загрузит чарты и подготовит их к установке вместе с вашим собственным приложением.

Развертывание чартов Helm

Далее поговорим о том, что нужно для развертывания приложения из чарта Helm. Одной из самых важных особенностей Helm является возможность создавать, изменять, обновлять и перезаписывать параметры конфигурации. В этом разделе мы посмотрим, как все работает.

Задание переменных

Как вы уже знаете, при создании чарта Helm можно разместить все параметры, подлежащие изменению со стороны пользователя, в файл `values.yaml` и дать им значения по умолчанию. Каким же образом *пользователь* чарта может поменять или перезаписать эти параметры, чтобы они подходили для его среды? Команда `helm install` позволяет указать файлы с дополнительными значениями, которые перезапишут то, что находится в `values.yaml`. Рассмотрим на примере.

Создание переменной среды

Возьмем ситуацию, когда вам нужно развернуть версию приложения в среде для финального тестирования. Допустим, на основании значения переменной среды под названием `ENVIRONMENT` приложение знает, находится оно в стадии подготовки

или производства, и соответственно меняет свое поведение (хотя в контексте нашего примера не так уж и важно, что это означает на практике). Как создается переменная среды?

Если вернуться к файлу `deployment.yaml`, можно заметить, что она поставляется в контейнер с помощью такого кода:

```
...
env:
  - name: ENVIRONMENT
    value: {{ .Values.environment }}
```

Вполне ожидаемо, что значение `environment` берется из `values.yaml`:

```
environment: development
...
```

Итак, если установить чарт со значениями по умолчанию, переменная `ENVIRONMENT` в контейнере будет равна `development`. Предположим, нам надо изменить ее на `staging`. Можно просто отредактировать файл `values.yaml`, как мы ранее делали, но есть лучший способ — создать новый файл YAML, содержащий значение этой одной переменной:

```
environment: staging
```

Переменную можно найти в файле `k8s/demo/staging-values.yaml`, который не входит в чарт Helm — он там лишь для того, чтобы вы не набирали лишний текст.

Задание значений в выпуске Helm

Чтобы указать файл с дополнительными значениями для команды `helm install`, используйте флаг `--values`, как показано ниже:

```
helm install --name demo-staging --values=./k8s/demo/staging-values.yaml
./k8s/demo ...
```

Это создаст новый выпуск с новым именем (`demo-staging`), а переменной `ENVIRONMENT` в активном контейнере будет присвоено значение `staging` вместо `development`. Переменные, перечисленные в дополнительных файлах, которые мы указали с помощью `--values`, объединятся со значениями по умолчанию (из файла `values.yaml`). В данном случае имеется лишь одна переменная (`environment`) и `staging-values.yaml` переопределяет ее значение по умолчанию.

Команде `helm install` можно указывать значения прямо в командной строке, используя флаг `--set`, но это не в духе концепции «инфраструктура как код». Лучшим решением будет изменить параметры чарта Helm, создать файл YAML, который переопределит значения по умолчанию (как это делает файл `staging-`

`values.yaml` в нашем примере), и применить его в командной строке с помощью флага `--values`.

Само собой, что этот способ следует использовать для задания конфигурационных значений при установке собственных чартов Helm, но то же самое можно делать и с публичными чартами. Чтобы просмотреть список значений, доступных для изменения, выполните команду `helm inspect values`, указав имя чарта:

```
helm inspect values stable/prometheus
```

Обновление приложения с помощью Helm

Вы уже знаете, как устанавливать чарты Helm со значениями по умолчанию и как переопределять эти значения с помощью пользовательских файлов. Но как изменить параметры уже запущенного приложения?

Для этого существует команда `helm upgrade`. Представьте, что вам нужно изменить количество реплик (копий pod-оболочки, которые должны работать в Kubernetes) для демонстрационного приложения. Как можно увидеть в файле `values.yaml`, по умолчанию оно равно `1`:

```
replicas: 1
```

Переопределить можно с помощью пользовательского файла, поэтому отредактируйте `staging-values.yaml`, добавив соответствующую строку:

```
environment: staging
replicas: 2
```

Выполните следующую команду, чтобы применить изменения к *существующему* развертыванию `demo-staging`, не создавая нового:

```
helm upgrade demo-staging --values=./k8s/demo/staging-values.yaml ./k8s/demo
Release "demo-staging" has been upgraded. Happy Helming!
```

Команду `helm upgrade` можно выполнять столько раз, сколько нужно, обновляя активное развертывание, — Helm не станет возражать.

Откат к предыдущей версии

Если вы решите, что вам не нравится версия, которую вы только что развернули, или возникли какие-то проблемы, легко можно вернуться к предыдущей версии с помощью команды `helm rollback`, просто указав номер предыдущего выпуска (его можно узнать в выводе команды `helm history`):

```
helm rollback demo-staging 1
Rollback was a success! Happy Helming!
```

На самом деле откат не обязательно должен быть к предыдущему выпуску. Предположим, вы откатились к ревизии 1, а затем решили, что нужно перейти *вперед* к ревизии 2. Именно это позволяет сделать команда `helm rollback demo-staging 2`.

Автоматический откат с помощью helm-monitor

Helm позволяет выполнять автоматический откат на основе показателей (см. главу 16). Например, вы используете Helm в рамках процесса непрерывного развертывания (см. главу 14) и, вполне вероятно, хотите, чтобы обновление автоматически отменялось, когда количество ошибок, записанных вашей системой мониторинга, превысит определенное число.

Для этого воспользуйтесь дополнением `helm-monitor`, которое может обращаться к серверу Prometheus (см. подраздел «Prometheus» на с. 371) за любым нужным вам выражением со снятыми показателями. Оно инициирует откат, если обращение пройдет успешно. `helm-monitor` будет наблюдать за показателями на протяжении пяти минут и откатит выпуск, если за это время обнаружит проблему. Больше информации о `helm-monitor` вы найдете в этой статье: blog.container-solutions.com/automated-rollback-helm-releases-based-logs-metrics.

Создание репозитория с чартами Helm

До сих пор мы использовали Helm для установки чартов либо из локальной папки, либо из репозитория `stable`. Для работы с Helm вам не нужен собственный репозиторий, так как чарты часто хранятся в репозиториях самих приложений.

Но если вам все же потребуется свой репозиторий чартов Helm, создать его достаточно просто. Чарты должны быть доступными по HTTP, и добиться этого можно множеством различных способов: поместить их в бакет облачного хранилища, использовать GitHub Pages или собственный веб-сервер, если таковой у вас имеется.

Собрав все свои чарты в единой папке, перейдите в нее и выполните команду `helm repo index`, чтобы создать файл `index.yaml` с метаданными репозитория.

Вот и все: ваш репозиторий чартов готов к работе! Подробнее об этом можно почитать в документации Helm (helm.sh/docs/developing_charts/#the-chart-repository-guide).

Для установки чартов достаточно добавить свой репозиторий в список Helm:

```
helm repo add myrepo http://myrepo.example.com
helm install myrepo/myapp
```

Управление конфиденциальными данными чартов Helm с помощью Sops

В разделе «Конфиденциальные данные в Kubernetes» на с. 243 было показано, как хранить конфиденциальные данные в Kubernetes и как передавать их приложениям через переменные среды или подключаемые файлы. Если у вас есть два и более секретных значения, проще будет создать для них один файл, а не отдельные файлы для каждого значения. Используя Helm для развертывания приложения, можете разместить свои значения в файле `values.yaml` и зашифровать его с помощью Sops (см. раздел «Шифрование конфиденциальных данных с помощью Sops» на с. 252).

Мы подготовили для вас пример в папке `hello-sops` репозитория `demo`:

```
cd hello-sops
tree
.
├── k8s
│   └── demo
│       ├── Chart.yaml
│       ├── production-secrets.yaml
│       ├── production-values.yaml
│       ├── staging-secrets.yaml
│       ├── staging-values.yaml
│       ├── templates
│       │   ├── deployment.yaml
│       │   └── secrets.yaml
│       └── values.yaml
└── temp.yaml

3 directories, 9 files
```

Этот чарт Helm своей структурой похож на чарт, который был в одном из наших предыдущих примеров (см. подраздел «Что внутри у чарта Helm» на с. 281). Здесь мы определили ресурсы `Deployment` и `Secret`, слегка изменив их так, чтобы упростить управление большим количеством секретных значений в разных средах.

Посмотрим, какие конфиденциальные данные нужны нашему приложению:

```
cat k8s/demo/production-secrets.yaml
secret_one: ENC[AES256_GCM,data:ekH3xIdCFiS4j1I2ja8=,iv:C95KilXL...1g==,type:str]
secret_two: ENC[AES256_GCM,data:0Xcmm1cdv3TbfM3mIkA=,iv:PQOcI9vX...XQ==,type:str]
...
```

Мы использовали Sops для шифрования значений, которые будут использоваться приложением.

Теперь взгляните на файл `secrets.yaml`:

```
cat k8s/demo/templates/secrets.yaml
apiVersion: v1
kind: Secret
metadata:
  name: {{ .Values.container.name }}-secrets
type: Opaque
data:
    {{ $environment := .Values.environment }}
    app_secrets.yaml: {{ .Files.Get (nospace (cat $environment "-secrets.yaml"))
        | b64enc }}
```

В последние две строчки мы добавили шаблонизацию Go, благодаря которой чарт Helm сможет читать конфиденциальные данные либо из `production-secrets.yaml`, либо из `staging-secrets.yaml` — в зависимости от значения переменной `environment`, указанной в файле `values.yaml`.

Конечным результатом будет единый манифест объекта `Secret` под названием `app_secrets.yaml`, содержащий *все* пары «ключ — значение», которые определены в конфиденциальных файлах. Этот объект будет подключен к развертыванию в виде единого файла для использования приложением.

Мы также добавили `...| b64enc` в конец последней строчки. Это еще одна полезная функция шаблонов Go в Helm, позволяющая автоматически преобразовывать конфиденциальные данные в виде простого текста в кодировку `base64`: по умолчанию именно такими Kubernetes ожидает увидеть эти данные (см. пункт «base64» на с. 246).

Сначала нам нужно временно расшифровать файлы с помощью Sops, а затем применить изменения к кластеру Kubernetes. Вот последовательность команд для развертывания финальной тестовой версии демонстрационного приложения с соответствующими секретными значениями:

```
sops -d k8s/demo/staging-secrets.yaml > temp-staging-secrets.yaml && \
helm upgrade --install staging-demo --values staging-values.yaml \
--values temp-staging-secrets.yaml ./k8s/demo && rm temp-staging-secrets.yaml
```

Это работает следующим образом.

1. Sops расшифровывает файл `staging-secrets` и записывает результат в `temp-staging-secrets`.
2. Helm устанавливает чарт `demo`, используя значения из `staging-values` и `temp-staging-secrets`.
3. Файл `temp-staging-secrets` удаляется.

Поскольку все это происходит в один этап, мы не оставляем расшифрованный файл с конфиденциальными данными, который могут найти не те люди.

Управление несколькими чартами с помощью Helmfile

При знакомстве с Helm в разделе «Helm: диспетчер пакетов для Kubernetes» на с. 102 мы показали, как развернуть чарт с демонстрационным приложением в кластере Kubernetes. Но, несмотря на все свои достоинства, Helm одновременно работает лишь с одним чартом. Как узнать, какие приложения должны выполняться в вашем кластере? Помните, что во время их установки с помощью Helm к ним применяются пользовательские настройки.

С этим вам может помочь удобный инструмент под названием Helmfile (github.com/roboll/helmfile).

По аналогии с тем, как Helm позволяет развернуть отдельное приложение с использованием шаблонизации и переменных, Helmfile делает возможным развертывание всего, что должно быть установлено в вашем кластере, с помощью одной-единственной команды.

Что такое Helmfile

В репозитории `demo` находится пример того, как использовать Helmfile. В папке `hello-helmfile` вы найдете файл `helmfile.yaml`:

```
repositories:
  - name: stable
    url: https://kubernetes-charts.storage.googleapis.com/

releases:
  - name: demo
    namespace: demo
    chart: ../hello-helm/k8s/demo
    values:
      - "../hello-helm/k8s/demo/production-values.yaml"

  - name: kube-state-metrics
    namespace: kube-state-metrics
    chart: stable/kube-state-metrics

  - name: prometheus
    namespace: prometheus
    chart: stable/prometheus
    set:
      - name: rbac.create
        value: true
```

В разделе `repositories` определяются репозитории с чартами Helm, на которые мы будем ссылаться. В данном случае мы указываем лишь `stable` — официальный

стабильный репозиторий чартов для Kubernetes. Если вы храните чарты локально (см. подраздел «Создание репозитория с чартами Helm» на с. 288), добавьте сюда свой репозиторий.

Дальше в разделе `releases` определяется набор выпусков: приложений, которые мы бы хотели развернуть в кластере. Для каждого выпуска указываются некоторые из следующих метаданных:

- `name`. Имя чарта Helm, который нужно развернуть;
- `namespace`. Пространство имен для развертывания;
- `chart`. URL или путь к самому чарту;
- `values`. Путь к файлу `values.yaml`, который будет использоваться вместе с развертыванием;
- `set`. Любые дополнительные значения, помимо тех, что указаны в файле `values.yaml`.

Мы определили здесь три выпуска: демонстрационное приложение, Prometheus (см. подраздел «Prometheus» на с. 371) и `kube-state-metrics` (см. подраздел «Показатели Kubernetes» на с. 353).

Метаданные чарта

Обратите внимание на то, что мы указали относительный путь к чарту `demo` и файлам со значениями:

```
- name: demo
  namespace: demo
  chart: ../hello-helm/k8s/demo
  values:
    - "../hello-helm/k8s/demo/production-values.yaml"
```

Таким образом Helmfile работает с чартами, которые не находятся в репозитории Helm: вы можете хранить их все, например, в одном и том же репозитории исходного кода.

Для чарта `prometheus` мы указали значение `stable/prometheus`. Поскольку это не путь файловой системы, Helmfile знает, что чарт следует искать в репозитории `stable`, который мы определили ранее в разделе `repositories`:

```
- name: stable
  url: https://kubernetes-charts.storage.googleapis.com/
```

У всех чартов есть различные наборы значений по умолчанию, хранящиеся в их файлах `values.yaml`. В разделе `set:` манифеста Helmfile можно указать любые значения, которые вам бы хотелось перезаписать при установке приложения.

В этом примере мы поменяли в выпуске `prometheus` значение `rbac.create` с `true` на `false`:

```
- name: prometheus
  namespace: prometheus
  chart: stable/prometheus
  set:
    - name: rbac.create
      value: true
```

Применение Helmfile

Файл `helmfile.yaml` декларативно описывает все, что должно выполняться в кластере (по крайней мере какую-то совокупность действий), как это делается в манифестах Kubernetes. Если применить декларативный манифест, Helmfile приведет кластер в соответствие с вашей спецификацией.

Выполните следующее:

```
helmfile sync
exec: helm repo add stable https://kubernetes-charts.storage.googleapis.com/
"stable" has been added to your repositories
exec: helm repo update
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "cloudnativedevops" chart repository
...Successfully got an update from the "stable" chart repository
Update Complete. ⎈ Happy Helming!⎈
exec: helm dependency update .../demo/hello-helm/k8s/demo
...
```

Это то же самое, как выполнить команды `helm install` / `helm upgrade` для каждого чарта Helm, который вы определили.

Если вы хотите выполнять `helm sync` автоматически — например, в рамках процесса непрерывного развертывания (см. главу 14), то вместо того, чтобы добавлять приложение в кластер с помощью `helm install`, можно просто отредактировать файл Helmfile, поместить его в систему контроля версий и дождаться, когда система автоматизации выкатит ваши изменения.

Используйте единый источник достоверных данных. Не смешивайте ручное развертывание отдельных чартов с помощью Helm и декларативное управление всеми чартами кластера посредством Helmfile: выберите что-то одно. Если вы примените Helmfile, а затем воспользуетесь Helm для развертывания или модификации приложений вручную, Helmfile перестанет быть единым источником достоверных данных в вашем кластере, что обязательно приведет к проблемам. Поэтому, если вы используете Helmfile, убедитесь в том, что все ваши развертывания производятся именно им.

Если Helmfile вам не по душе, можете попробовать другие инструменты, которые делают примерно то же самое:

- Landscaper (github.com/Eneco/landscaper);
- Helmsman (github.com/Praqma/helmsman).

Как обычно, говоря о каких-либо инструментах, мы советуем почитать документацию, сравнить разные варианты, попробовать их в деле и затем решить, что вам больше подходит.

Продвинутые инструменты управления манифестами

Helm является отличным и широко используемым инструментом, но имеет несколько недостатков: написание и редактирование его шаблонов — не самое приятное занятие, а файлы YAML в Kubernetes довольно сложные, объемные и повторяющиеся.

Сейчас в разработке находится несколько инструментов, которые пытаются решить данные проблемы и упростить работу с манифестами Kubernetes: либо с помощью более мощных по сравнению с YAML языков описания, таких как Jsonnet, либо путем объединения манифестов в базовые шаблоны, которые затем настраиваются с помощью накладываемых файлов.

ksonnet

Иногда декларативных возможностей YAML недостаточно, особенно если речь идет о больших и сложных развертываниях, в которых нужно использовать вычисления и логику. Например, вы можете устанавливать количество реплик ди-

намически, в зависимости от размера кластера. Для этого нужен настоящий язык программирования.

> Файлы YAML в Kubernetes никогда не предназначались для ручного редактирования. ksonnet стремится предоставить единый инструмент, который облегчает настройку приложений Kubernetes в разных кластерах и средах.
>
> *Джо Беда (Heptio)*[1]

ksonnet (ksonnet.io) позволяет писать манифесты Kubernetes на языке под названием Jsonnet, который является расширенной версией JSON — декларативного формата данных, аналогичного YAML. Манифесты в формате JSON также поддерживаются в Kubernetes. Jsonnet расширяет возможности JSON: позволяет работать с переменными, циклами, условными выражениями, выполнять арифметические операции, обрабатывать ошибки и т. д.

```
local env = std.extVar("__ksonnet/environments");
local params = std.extVar("__ksonnet/params").components.demo;
[
   {
      "apiVersion": "v1",
      "kind": "Service",
      "metadata": {
      "name": params.name
   },
   "spec": {
      "ports": [
         {
            "port": params.servicePort,
            "targetPort": params.containerPort
            ...
```

Но самое важное то, что ksonnet вводит концепцию *прототипов*: заранее подготовленных наборов ресурсов Kubernetes, с помощью которых можно «штамповать» часто используемые шаблоны манифестов.

Например, один из встроенных в ksonnet прототипов — `deployed-service` — генерирует развертывание для заданного контейнера, а также сервис, который направляет к нему трафик. Это хорошая отправная точка для большинства приложений, которые запускаются в Kubernetes.

Вам доступна библиотека публичных прототипов, но вы можете определять собственные, чтобы не дублировать большое количество кода манифестов во всех своих приложениях и сервисах.

---

[1] blog.heptio.com/the-next-chapter-for-ksonnet-1dcbbad30cb.

kapitan

kapitan (github.com/deepmind/kapitan) — это еще один инструмент на основе Jsonnet, предназначенный для использования общих конфигурационных значений в разных приложениях или даже кластерах. У kapitan есть иерархическая база данных параметров (под названием *«инвентарь»*), которая позволяет повторно использовать шаблоны манифестов, подключая разные значения в зависимости от среды или приложения:

```
local kube = import "lib/kube.libjsonnet";
local kap = import "lib/kapitan.libjsonnet";
local inventory = kap.inventory();
local p = inventory.parameters;

{
    "00_namespace": kube.Namespace(p.namespace),
    "10_serviceaccount": kube.ServiceAccount("default")
}
```

kustomize

kustomize (github.com/kubernetes-sigs/kustomize) — это еще один инструмент для работы с манифестами, который использует стандартный формат YAML вместо шаблонов или альтернативных языков наподобие Jsonnet. Вы начинаете с базового манифеста, на который *накладываются* дополнительные слои, чтобы подогнать его под разные среды или конфигурации. Утилита командной строки `kustomize` берет все эти файлы и генерирует из них итоговые манифесты:

```
namePrefix: stagingcommonLabels:
  environment: staging
  org: acmeCorporation
commonAnnotations:
  note: Hello, I am staging!
bases:
- ../../base
patchesStrategicMerge:
- map.yaml
EOF
```

Благодаря этому развертывание манифестов можно свести к такой простой команде:

```
kustomize build /myApp/overlays/stagingq | kubectl apply -f -
```

Если вам не нравятся шаблоны или Jsonnet и вы хотите работать с обычными манифестами Kubernetes, этот инструмент заслуживает вашего внимания.

kompose

Скорее всего, вы знакомы с Docker Compose, если запускали промышленные сервисы в контейнерах Docker, но без использования Kubernetes.

Compose позволяет определить и развернуть группу контейнеров, работающих вместе: например, веб-сервер, серверное приложение и базу данных, такую как Redis. То, как эти контейнеры будут взаимодействовать между собой, можно описать с помощью единственного файла `docker-compose.yml`.

kompose (github.com/kubernetes/kompose) — инструмент для преобразования `docker-compose.yml` в манифесты Kubernetes. Он позволяет мигрировать из Docker Compose без необходимости писать с нуля собственные манифесты Kubernetes или чарты Helm.

Ansible

Возможно, вы уже знакомы с Ansible, популярным инструментом для автоматизации инфраструктуры. Он не имеет непосредственного отношения к Kubernetes, но способен управлять разными видами ресурсов, используя модули расширений, такие как Puppet (см. подраздел «Модуль Puppet для Kubernetes» на с. 81).

Кроме установки и настройки кластеров, Ansible умеет управлять такими ресурсами Kubernetes, как развертывания и сервисы: для этого используется модуль k8s (docs.ansible.com/ansible/latest/modules/k8s_module.html).

Как и Helm, Ansible способен шаблонизировать манифесты Kubernetes с помощью стандартного языка шаблонов (Jinja), предоставляя более развитый — иерархический — механизм поиска переменных. Например, вы можете задать общие значения для группы приложений или среды развертывания, такой как `staging`.

Даже если вы уже применяете Ansible в своей организации, стоит хорошо подумать, прежде чем использовать его и для работы с ресурсами Kubernetes. Для инфраструктуры, основанной исключительно на Kubernetes, Ansible может оказаться слишком мощным. Но в случае смешанной инфраструктуры вполне удобно иметь один инструмент для управления всем и вся:

```
kube_resource_configmaps:
  my-resource-env: "{{ lookup('template', template_dir +
'/my-resource-env.j2') }}"
kube_resource_manifest_files: "{{ lookup('fileglob', template_dir +
'/*manifest.yml') }}"
- hosts: "{{ application }}-{{ env }}-runner"
  roles:
    - kube-resource
```

Возможность управления Kubernetes с помощью Ansible (willthames.github.io/ansiblefest2018/#/) рассматривается в докладе Уилла Тэймса, специализирующегося на этой теме.

kubeval

В отличие от других инструментов, на которые мы обратили внимание в этом разделе, kubeval (github.com/instrumenta/kubeval) предназначен не для генерации или шаблонизации манифестов Kubernetes, а для их проверки.

У каждой версии Kubernetes есть своя схема манифестов в форматах YAML и JSON, поэтому важно иметь возможность автоматически проверять манифесты на соответствие этой схеме. Например, kubeval помогает удостовериться, все ли обязательные поля вы указали для заданного объекта и корректные ли типы у значений.

И `kubectl` проверяет манифесты во время их применения, выдавая сообщения об ошибках, когда вы пытаетесь применить некорректный манифест. Но возможность сделать это предварительно тоже может быть полезной. kubeval не требует доступа к кластеру и поддерживает любые версии Kubernetes.

kubeval лучше использовать в рамках процесса непрерывного развертывания, чтобы манифесты проверялись всякий раз, когда вы вносите в них изменения. kubeval также позволяет, к примеру, перед обновлением до последней версии Kubernetes проверить, требуют ли ваши манифесты какой-либо модификации для работы с ней.

Резюме

Развертывание приложений в Kubernetes можно выполнять с помощью обычных манифестов YAML, но это неудобно. Более комфортно пользоваться таким мощным инструментом, как Helm, однако при условии, что вы знаете, как извлечь из него максимальную выгоду.

В настоящее время разрабатывается множество инструментов, нацеленных на то, чтобы сделать развертывание в Kubernetes более простым. Некоторые из их функций будут интегрированы с Helm. В любом случае основы работы с Helm стоит знать.

- Чарт — это спецификация пакета Helm, содержащая метаданные о пакете, некоторые из его конфигурационных значений и шаблоны объектов Kubernetes, которые на эти значения ссылаются.

- В результате установки чарта создается выпуск Helm. Каждый раз, когда вы устанавливаете экземпляр чарта, создается новый выпуск. Обновляя выпуск с помощью других конфигурационных значений, Helm инкрементирует номер ревизии выпуска.
- Чтобы подстроить чарт Helm под свои нужды, создайте файл с пользовательскими значениями, переопределяющими нужные вам параметры, и укажите его в командной строке `helm install` или `helm upgrade`.
- Вы можете использовать переменную (например, `environment`) для выбора разных наборов значений или конфиденциальных данных в зависимости от среды развертывания: `staging`, `production` и т. д.
- Helmfile позволяет декларативно описать набор чартов и значений, которые будут применены к вашему кластеру, и установить или обновить их все с помощью единственной команды.
- Helm можно использовать вместе с Sops для хранения секретной конфигурации в ваших чартах. При этом можно указать функцию, которая автоматически переведет ваши конфиденциальные данные в кодировку base64, ожидаемую Kubernetes.
- Helm — не единственный инструмент, позволяющий управлять манифестами Kubernetes. ksonnet и kapitan используют Jsonnet, альтернативный язык шаблонов. kustomize предлагает иной подход: вместо интерполяции переменных используется наложение друг на друга файлов YAML, которые модифицируют манифесты.
- kustomize позволяет быстро проверить синтаксис манифестов и найти типичные ошибки.

13 Процесс разработки

> Серфинг — это такая потрясающая концепция! Вы со своей маленькой доской бросаете вызов Природе: «Я тебя оседлаю!» А Природа часто отвечает: «Ну уж нет!» — и швыряет тебя на дно.
>
> *Джолин Блэлок*

Здесь мы продолжим обсуждение, начатое в главе 12, но сосредоточимся на жизненном цикле приложения: от локальной разработки до развертывания обновлений в кластере Kubernetes, и в том числе затронем такую непростую тему, как миграция баз данных. Рассмотрим также определенные инструменты, которые помогут вам разрабатывать, тестировать и развертывать приложения: Skaffold, Draft, Telepresence и Knative. Когда вы будете готовы к размещению своего приложения в кластере, мы расскажем и о более сложных способах развертывания с использованием хуков Helm.

Инструменты разработки

В главе 12 мы рассмотрели разные инструменты, которые могут помочь с написанием, сборкой и развертыванием манифестов ресурсов Kubernetes. Эти инструменты, несомненно, полезны, но часто при разработке приложения, выполняющегося в Kubernetes, хочется иметь возможность опробовать и сразу увидеть вносимые изменения, еще до окончания полного цикла построения — загрузки — развертывания — обновления.

Skaffold

Skaffold (github.com/GoogleContainerTools/skaffold) — это инструмент с открытым исходным кодом от Google, предназначенный для организации быстрого процесса локальной разработки. Он автоматически пересобирает ваши контейнеры по мере того, как вы пишете код у себя на компьютере, и развертывает эти изменения в локальном или удаленном кластере.

Нужный вам процесс описывается в файле `skaffold.yaml` репозитория, а затем запускается с помощью утилиты командной строки `skaffold`. Когда вы изменяете файлы в своей локальной папке, Skaffold пробуждается, собирает новый контейнер и автоматически его развертывает. Это экономит время на загрузку контейнера в реестр.

Draft

Draft (github.com/Azure/draft) — инструмент с открытым исходным кодом, разрабатываемый командой Microsoft Azure. Как и Skaffold, он умеет использовать Helm для автоматического развертывания обновлений в кластере при изменении кода.

Draft также имеет собственный механизм пакетов: готовых файлов Dockerfile и чартов Helm для того языка, который вы используете. На сегодняшний день эти пакеты доступны для .NET, Go, Node, Erlang, Clojure, C#, PHP, Java, Python, Rust, Swift и Ruby.

Если вы только начинаете писать новое приложение и у вас еще нет Dockerfile или чарта Helm, Draft может быть идеальным решением для того, чтобы быстро приступить к работе. Если выполнить `draft init && draft create`, Draft проанализирует файлы в локальной папке вашего приложения, попытается определить, на каком языке написан код, и затем создаст соответствующие Dockerfile и чарт Helm.

Чтобы все это применить, введите команду `draft up`. Draft соберет локальный контейнер Docker, используя созданный им экземпляр Dockerfile, и развернет его в вашем кластере Kubernetes.

Telepresence

Telepresence (www.telepresence.io) применяет немного другой подход по сравнению со Skaffold и Draft. Вам не нужен локальный кластер Kubernetes: pod-оболочка Telepresence работает в вашем настоящем кластере и служит заглушкой для приложения. Трафик, который предназначается вам, перехватывается и направляется к контейнеру, работающему на вашем локальном компьютере.

Благодаря этому локальный компьютер разработчика может стать участником удаленного кластера. Изменения, вносимые в код приложения, будут доступны в реальных условиях без необходимости развертывать новый контейнер.

Knative

Если другие рассмотренные нами инструменты концентрируются на ускорении цикла локальной разработки, то проект Knative (github.com/knative/docs) более амбициозен. Он пытается предоставить стандартный механизм развертывания в Kubernetes всех видов рабочих заданий: не только централизованных приложений, но и функций в *бессерверном* стиле.

Knative интегрируется сразу с Kubernetes и Istio (см. раздел «Istio» на с. 230) и предоставляет полноценную платформу развертывания приложений/функций, включая конфигурацию процесса сборки, автоматическую доставку кода и *событийный* механизм, который стандартизует то, как приложения используют системы очередей и обмена сообщениями (такие как Pub/Sub, Kafka или RabbitMQ).

Проект Knative находится на ранней стадии своего развития, но за ним стоит понаблюдать.

Стратегии развертывания

Если вы хотите обновить приложение вручную (без участия Kubernetes), сначала закройте его, затем установите новую версию и запустите. Но это означает, что какое-то время ваше приложение будет недоступно.

При наличии нескольких реплик их лучше обновлять по очереди, чтобы не допускать прерывания в обслуживании: это так называемое *развертывание без простоя*.

Нулевое время простоя требуется не всем приложениям: например, внутренние сервисы, которые потребляют очереди сообщений, являются идемпотентными, поэтому их можно обновить за один раз. Это означает, что обновление выполняется быстрее. Но для приложений, с которыми работают пользователи, обычно более важным фактором является постоянная доступность.

В Kubernetes вы можете выбрать наиболее подходящую вам стратегию. `RollingUpdate` исключает простаивание и обновляет pod-оболочки одну за другой, тогда как `Recreate` позволяет работать быстрее и со всеми pod-оболочками одновременно. Есть также поля, подкорректировав которые можно добиться поведения, необходимого вашему приложению.

В Kubernetes стратегия развертывания приложения определяется в манифесте ресурса `Deployment`. По умолчанию используется `RollingUpdate`, поэтому, если все оставить как есть, вы получите именно ее. Поменять на `Recreate` можно так:

```
apiVersion: extensions/v1beta1
kind: Deployment
spec:
  replicas: 1
  strategy:
    type: Recreate
```

Далее более подробно рассмотрим эти стратегии и то, как они работают.

Плавающие обновления RollingUpdate

Если выбрать `RollingUpdate`, pod-оболочки будут обновляться по очереди до тех пор, пока все реплики не перейдут на новую версию.

Представьте, например, что у приложения есть три реплики, каждая с версией `v1`. Разработчик начинает переход на версию `v2`, используя команды `kubectl apply...` или `helm upgrade...`. Что произойдет?

Вначале одна из трех pod-оболочек версии `v1` будет удалена. Kubernetes пометит ее как неготовую к работе и перестанет отправлять ей трафик. Вместо нее запустится новая pod-оболочка версии `v2`. Тем временем оставшиеся оболочки версии `v1` по-прежнему будут получать входящие запросы. Пока замена готовится к работе, количество pod-оболочек уменьшится до двух, но обслуживание пользователей продолжится.

Когда `Pod`-объект `v2` будет готов, Kubernetes начнет направлять ему пользовательский трафик, так же как и двум другим репликам. Таким образом, мы вернемся к нашему полному комплекту из трех pod-оболочек.

Этот процесс с `Pod`-объектами продолжится до тех пор, пока все реплики версии `v1` не будут заменены репликами версии `v2`. Таким образом, в определенные моменты времени для обработки трафика доступно меньшее число pod-оболочек, но в целом приложение никогда не простаивает. Это развертывание без простоя.

Во время плавающего обновления старая и новая версии вашего приложения будут работать параллельно. Обычно это не вызывает проблем, но, возможно, придется предпринять некоторые шаги, чтобы сделать этот процесс безопасным. Например, если ваши изменения влекут за собой миграцию базы данных (см. раздел «Выполнение миграции с помощью Helm» на с. 307), нормальное плавающее обновление невозможно.

Если вскоре после перехода в состояние готовности ваши pod-оболочки начинают отказывать, используйте поле `minReadySeconds`, чтобы обновление не запускалось, пока они не стабилизируются (см. подраздел «Поле minReadySeconds» на с. 115).

Стратегия Recreate

В режиме `Recreate` все запущенные реплики удаляются одновременно, а потом создаются новые.

Это вполне приемлемо для приложений, которые не обрабатывают запросы напрямую. Одним из преимуществ данной стратегии является то, что она исключает ситуацию с параллельной работой двух разных версий одного приложения (см. подраздел «Плавающие обновления RollingUpdate» на с. 303).

Параметры maxSurge и maxUnavailable

Иногда, пока выкатывается обновление, у вас может оказаться больше или меньше pod-оболочек, чем указано в поле `replicas`. Такое поведение регулируется двумя важными параметрами: `maxSurge` и `maxUnavailable`:

- `maxSurge` задает максимальное количество лишних pod-оболочек. Например, если у вас есть десять реплик, а параметр `maxSurge` равен 30 %, система не допустит, чтобы число одновременно запущенных `Pod`-объектов превысило 13;
- `maxUnavailable` задает максимальное количество недоступных pod-оболочек. Если номинальное число реплик равно десяти, а параметр `maxUnavailable` равен 20 %, Kubernetes никогда не позволит, чтобы количество доступных `Pod`-объектов опустилось ниже восьми.

Эти значения можно устанавливать в виде целых чисел или процентов:

```
apiVersion: extensions/v1beta1
kind: Deployment
spec:
  replicas: 10
  strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 20%
    maxUnavailable: 3
```

Обычно значения по умолчанию для обоих параметров (25 % или 1, в зависимости от версии Kubernetes) вполне подходят и менять их нет необходимости. Но в некоторых ситуациях их следует откорректировать таким образом, чтобы ваше приложение могло поддерживать приемлемую пропускную способность во

время обновлений. При очень крупных масштабах может обнаружиться, что 75 % доступности недостаточно, и тогда придется немного уменьшить `maxUnavailable`.

Чем больше значение `maxSurge`, тем быстрее происходит обновление, но и тем более нагружены ресурсы вашего кластера. Большие значения `maxUnavailable` тоже ускоряют выкатывание, но за счет пропускной способности приложения.

С другой стороны, маленькие значения `maxSurge` и `maxUnavailable` уменьшают влияние на ваш кластер и пользователей, но при этом обновление может существенно замедлиться. Выбор правильного компромисса ложится на вас.

Сине-зеленые развертывания

Когда происходит *сине-зеленое* обновление, заменяются не отдельные `Pod`-объекты, а целое развертывание: создается новый объект `Deployment` с самостоятельным стеком pod-оболочек версии `v2`, который запускается рядом с существующим развертыванием версии `v1`.

Одно из преимуществ такого подхода в том, что вам не нужно иметь дело со старыми и новыми версиями приложения, которые обслуживают запросы одновременно. С другой стороны, ваш кластер должен быть достаточно большим, чтобы в нем можно было удвоить количество реплик, необходимых вашему приложению, — а это может быть дорого. К тому же большую часть времени у вас без дела будет находиться неиспользуемая емкость (разве что вы масштабируете кластер по мере необходимости).

Как вы помните из подраздела «Ресурсы типа “сервис”» на с. 97, для определения того, какие pod-оболочки должны получать трафик от сервиса, Kubernetes использует метки. Чтобы реализовать сине-зеленое развертывание, старые и новые `Pod`-объекты стоит помечать по-разному (см. раздел «Метки» на с. 205).

Если немного подправить определение сервиса в нашем демонстрационном приложении, трафик можно направлять только pod-оболочкам с меткой `deployment: blue`:

```
apiVersion: v1
kind: Service
metadata:
  name: demo
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
  app: demo
  deployment: blue
type: ClusterIP
```

При развертывании новой версии вы можете пометить ее как `deployment: green`. Она не получит никакого трафика даже в состоянии полной готовности, так как сервис шлет запросы только pod-оболочкам с меткой `blue`. Прежде чем выполнять переход, обновление можно проверить и убедиться в том, что оно готово к работе.

Чтобы переключиться на новое развертывание, отредактируйте сервис, поменяв селектор на `deployment: green`. Теперь pod-оболочки с меткой `green` станут получать трафик, и вы избавитесь от старых реплик, когда они начнут простаивать.

Rainbow-развертывания

В редких случаях, особенно когда pod-оболочки имеют очень продолжительные соединения (например, веб-сокеты), сине-зеленых развертываний может быть недостаточно. Иногда приходится поддерживать одновременно три активные версии приложения и больше.

Такой подход иногда называют *rainbow-развертыванием*: при каждом обновлении вы получаете набор pod-оболочек другого цвета. Дальше вы можете останавливать старые версии по мере того, как они закрывают свои соединения.

Подробно пример rainbow-развертывания (github.com/bdimcheff/rainbow-deploys) описывает Брэндон Димчефф.

Канареечные развертывания

Преимущество сине-зеленых (или rainbow) развертываний состоит в том, что в случае, если обновление ведет себя некорректно или чем-то вас не устраивает, вы можете просто вернуться к запущенной старой версии. Однако это довольно накладно, поскольку вам нужны ресурсы для одновременного выполнения сразу двух версий.

Альтернативным подходом, позволяющим избежать данной проблемы, является *канареечное развертывание*. Как и канарейка в угольной шахте, новые pod-оболочки подвергаются всем опасностям промышленной среды, чтобы была возможность посмотреть, как они себя поведут. Если pod-оболочки «выживают», выкатывание можно продолжать до полного завершения. Но если проблема все же *возникает*, ее масштаб строго ограничен.

Как и в случае с сине-зелеными развертываниями, реализовать это можно с помощью меток (см. раздел «Метки» на с. 205). Подробный пример канареечного обновления представлен в документации Kubernetes (kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#canary-deployments).

Существует и более сложный способ, основанный на использовании Istio (см. раздел «Istio» на с. 230). Он позволяет перенаправлять колеблющуюся долю трафика к одной или нескольким версиям сервиса, выбранным произвольным образом. Это упрощает выполнение таких процедур, как A/B-тестирование.

Выполнение миграции с помощью Helm

Приложения, не хранящие свое состояние, легко развертываются и обновляются, но если в этом процессе участвует база данных, все может усложниться. Изменение схемы хранилища обычно требует выполнения *миграции* на одном из этапов выкатывания. Например, для приложений на основе Rails при старте новых pod-оболочек необходимо выполнить `rake db:migrate`.

В Kubernetes для этого можно использовать ресурс `Job` (см. подраздел «Запланированные задания» на с. 220): оформить скрипт с помощью команды `kubectl` в рамках процедуры обновления или, если вы работаете с Helm, прибегнуть к встроенной возможности под названием *«хуки»*.

Хуки Helm

Хуки Helm позволяют контролировать порядок, в котором выполняются этапы развертывания. С их помощью также можно отменить обновление, если что-то пойдет не так.

Вот пример задания по миграции базы данных для приложения Rails, развернутого с помощью Helm:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: {{ .Values.appName }}-db-migrate
  annotations:
    "helm.sh/hook": pre-upgrade
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  activeDeadlineSeconds: 60
  template:
      name: {{ .Values.appName }}-db-migrate
  spec:
    restartPolicy: Never
    containers:
    - name: {{ .Values.appName }}-migration-job
      image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
```

```
    command:
      - bundle
      - exec
      - rails
      - db:migrate
```

Свойства `helm.sh/hook` определены в разделе `annotations`:

```
annotations:
  "helm.sh/hook": pre-upgrade
  "helm.sh/hook-delete-policy": hook-succeeded
```

Параметр `pre-upgrade` заставляет Helm применить манифест этого задания перед выполнением обновления. Задание запустит стандартную команду миграции Rails.

Строчка `"helm.sh/hook-delete-policy": hook-succeeded` говорит Helm о том, что в случае успешного завершения (то есть при выходе со статусом 0) задание следует удалить.

Обработка неудачных хуков

Если задание возвращает ненулевой код завершения, значит, произошла ошибка и миграция не завершилась успешно. Helm оставит задание в состоянии сбоя, чтобы вы могли его отладить и узнать, что пошло не так.

В этом случае процесс выкатывания выпуска остановится и приложение не обновится. С помощью команды `kubectl get pods -a` можно увидеть список отказавших pod-оболочек, чтобы исследовать их журнальные записи и разобраться в произошедшем.

Как только проблема будет решена, вы можете удалить неудачное задание (`kubectl delete job <job-name>`) и запустить обновление еще раз.

Другие хуки

Помимо `pre-upgrade`, у хуков есть и другие этапы. Вы можете использовать хук на любой из этих стадий выпуска:

- `pre-install`. Выполняется после обработки шаблона, но перед созданием ресурсов;
- `post-install`. Выполняется после загрузки всех ресурсов;
- `pre-delete`. Выполняется при запросе на удаление, еще до удаления каких-либо ресурсов;

- `post-delete`. Выполняется при запросе на удаление уже после удаления всех ресурсов выпуска;
- `pre-upgrade`. Выполняется при запросе на обновление после обработки шаблонов, но перед загрузкой каких-либо ресурсов (например, перед операцией `kubectl apply`);
- `post-upgrade`. Выполняется при обновлении после обновления всех ресурсов;
- `pre-rollback`. Выполняется при запросе на откат после обработки шаблонов, но перед откатом каких-либо ресурсов;
- `post-rollback`. Выполняется при запросе на откат, после модификации всех ресурсов.

Создание цепочки хуков

Хуки Helm также можно объединять в упорядоченные цепочки, используя свойство `helm.sh/hook-weight`. Они будут выполняться в порядке возрастания, поэтому задание, у которого поле `hook-weight` равно `0`, запускается перед заданием с `hook-weight`, равным `1`:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: {{ .Values.appName }}-stage-0
  annotations:
    "helm.sh/hook": pre-upgrade
    "helm.sh/hook-delete-policy": hook-succeeded
    "helm.sh/hook-weight": "0
```

Все, что вам нужно знать о хуках, можно найти в документации Helm (helm.sh/docs/developing_charts/#hooks).

Резюме

Разработка приложений Kubernetes может быть утомительной, если для проверки малейшего изменения в коде вам придется собирать, загружать и развертывать образы контейнеров. Этот цикл и разработку в целом можно существенно ускорить с помощью таких инструментов, как Draft, Skaffold и Telepresence.

На практике Kubernetes значительно упрощает выкатывание изменений в промышленную среду по сравнению с традиционными серверами — при условии,

что вы понимаете основные концепции и умеете адаптировать их под свои приложения.

- Стратегия развертывания `RollingUpdate`, которая применяется в Kubernetes по умолчанию, обновляет одновременно лишь несколько pod-оболочек, останавливая старые версии только после того, как новые будут готовы к работе.
- Плавающие обновления позволяют избежать простоя, но при этом страдает скорость выкатывания. Это также означает, что во время выкатывания у вас будут параллельно работать как старые, так и новые версии приложения.
- Вы можете откорректировать плавающие обновления с помощью полей `maxSurge` и `maxUnavailable`. В зависимости от используемой вами версии API Kubernetes значения по умолчанию могут соответствовать или не соответствовать вашей ситуации.
- Стратегия `Recreate` просто убирает все старые pod-оболочки и одновременно с этим запускает их новые версии. Это быстрый процесс, но он приводит к простою, что не подходит для приложений, которые взаимодействуют с пользователями.
- При сине-зеленом развертывании все новые pod-оболочки запускаются и готовятся к работе, не получая никакого пользовательского трафика. Затем к ним перенаправляется весь трафик, еще до удаления старых версий.
- Rainbow-развертывания похожи на сине-зеленые, но допускают наличие более двух версий.
- В Kubernetes сине-зеленые развертывания и rainbow-развертывания можно реализовать путем назначения меток pod-оболочкам и изменения селектора в клиентском сервисе для направления трафика к нужному набору реплик.
- Хуки Helm позволяют применять определенные ресурсы Kubernetes (обычно запланированные задания) на той или иной стадии развертывания: например, чтобы выполнить миграцию базы данных.
- Хуки могут определять порядок, в котором ресурсы должны применяться в ходе развертывания, и останавливать выкатывание, если что-то пойдет не так.

14 Непрерывное развертывание в Kubernetes

> Дао ничего не делает, но нет того, что бы оно не сделало.
>
> *Лао-цзы*

В этой главе мы рассмотрим ключевой принцип DevOps — *непрерывное развертывание* и покажем, как его достичь в облачно-ориентированной среде, основанной на Kubernetes. Мы обсудим несколько способов организации процесса непрерывного развертывания с помощью Kubernetes и представим полностью рабочий пример с применением Cloud Build от Google.

Что такое непрерывное развертывание

Непрерывное развертывание (continuous deployment, CD) — это автоматическая доставка успешных сборок в промышленную среду. Как и пакет тестирования, развертывание должно быть централизованным и автоматизированным, а у разработчиков должна быть возможность развертывать новые версии либо нажатием кнопки, либо путем запроса на слияние, либо загружая в Git-репозиторий выпуск с определенной меткой.

Процесс CD часто связан с *непрерывной интеграцией* (continuous integration, CI), это значит с автоматической интеграцией и тестированием изменений, вносимых в магистральную ветку. Идея в том, что, если ваши изменения при слиянии с магистральной веткой нарушат сборку, процесс CI должен сразу же вам об этом сообщить, а не ждать, пока вы закончите работу с веткой и приступите

к заключительному слиянию. Сочетание непрерывных интеграции и развертывания обозначается так: *CI/CD*.

Механизм непрерывного развертывания часто называют *конвейером*: цепочкой автоматических действий по переносу кода с рабочей станции разработчика в промышленную среду с прохождением набора тестов и этапов приемки.

Типичный конвейер для контейнеризированного приложения выглядит следующим образом.

1. Разработчик загружает свои изменения в репозиторий Git.
2. Система сборки автоматически собирает текущую версию кода и выполняет тесты.
3. В случае прохождения всех тестов образ контейнера публикуется в центральном реестре.
4. Свежесобранный контейнер автоматически развертывается в среду заключительного тестирования.
5. Среда заключительного тестирования подвергается некоторым автоматическим приемочным проверкам.
6. Проверенный образ контейнера развертывается в промышленной среде.

Основная идея в том, что тестированию и развертыванию в различных средах подлежит не *исходный код*, а *контейнер*. Ошибки между исходным кодом и исполняемым файлом могут появиться множеством разных способов, и тестирование контейнера вместо кода выявит многие из них.

Огромным преимуществом CD является *отсутствие сюрпризов в промышленной среде*: развертывание происходит только после финального тестирования конкретного двоичного образа контейнера.

Подробный пример такого процесса CD приводится в разделе «Процесс CD с использованием Cloud Build» на с. 316.

Какие инструменты следует использовать для CD

Как обычно, проблема не в нехватке доступных инструментов, а в слишком широком их выборе. Существует несколько инструментов непрерывного развертывания, которые специально созданы для облачно-ориентированных приложений, а у давно устоявшихся традиционных систем, таких как Jenkins, есть дополнения, позволяющие работать с Kubernetes и контейнерами.

В итоге, если вы уже используете CD, вам, скорее всего, не придется переходить на совершенно другую систему: для перенесения в Kubernetes существующего приложения практически наверняка потребуется лишь слегка изменить процесс сборки.

Если вы еще не используете непрерывное развертывание, в этом разделе вам будет предложено несколько вариантов.

Jenkins

Jenkins (jenkins.io) — это широко используемый инструмент для CD, существующий уже много лет. У него есть дополнения практически для всего, что вам может понадобиться в рабочем процессе CD, включая Docker, `kubectl` и Helm.

Существует также более новый сторонний проект, JenkinsX (jenkins-x.io), предназначенный для запуска Jenkins в кластере Kubernetes.

Drone

Drone (github.com/drone/drone) — это относительно новый инструмент для CD, предназначенный для контейнеров и построенный с их помощью. Он простой и легковесный, а его рабочий процесс описывается в едином файле YAML. Поскольку каждый этап сборки состоит из запуска контейнера, это означает, что в Drone можно запускать любую контейнеризированную программу[1].

Google Cloud Build

Если ваша инфраструктура находится на платформе Google Cloud, при выборе системы для CD в первую очередь следует рассматривать Google Cloud Build. Как и Drone, Cloud Build запускает контейнеры на разных этапах сборки, а конфигурация в формате YAML размещается в репозитории вместе с вашим кодом.

Cloud Build можно настроить для наблюдения за вашим Git-репозиторием (доступна интеграция с GitHub). Когда выполняется заранее прописанное условие, такое как загрузка в определенную ветку или тег, Cloud Build запускает ваш процесс CD, собирая новый контейнер, выполняя набор тестов, публикуя образ и, возможно, развертывая новую версию в Kubernetes.

---

[1] Команда разработки New York Times написала полезную статью о развертывании в GKE с помощью Drone: open.blogs.nytimes.com/2017/01/12/continuous-deployment-to-google-cloud-platform-with-drone.

Полноценный рабочий пример процесса непрерывного развертывания в Cloud Build описан в разделе «Процесс CD с использованием Cloud Build» на с. 316.

Concourse

Concourse (concourse-ci.org) — это инструмент для CD с открытым исходным кодом, написанный на Go. Он тоже принимает декларативный подход к CD, подобно тому как это делают Drone и Cloud Build: для определения и выполнения этапов сборки используется файл в формате YAML. У Concourse уже есть официальный стабильный чарт Helm, который можно развернуть в Kubernetes. Это позволяет быстро и легко наладить контейнеризированный процесс CD.

Spinnaker

Spinnaker от Netflix — это очень мощный и гибкий инструмент, но на первый взгляд, слегка устрашающий. Его особенностью являются крупномасштабные и сложные развертывания, в том числе и сине-зеленые (см. подраздел «Сине-зеленые развертывания» на с. 305). Spinnaker посвящена бесплатная электронная книга (www.spinnaker.io/publications/ebook), которая поможет вам понять, подходит этот инструмент для ваших нужд или нет.

GitLab CI

GitLab — популярная альтернатива GitHub для размещения Git-репозиториев. В ее состав входит мощная система CD под названием GitLab CI (about.gitlab.com/product/continuous-integration), которую можно использовать для тестирования и развертывания кода.

Если вы уже используете GitLab, то стоит подумать о применении GitLab CI для реализации процесса непрерывного развертывания.

Codefresh

Codefresh (about.gitlab.com/product/continuous-integration) — это управляемый сервис CD для тестирования и развертывания приложений в Kubernetes. Одной из интересных его особенностей является возможность развертывать временные среды финального тестирования для каждой функциональной ветки.

CodeFresh может собирать, тестировать и развертывать нужные вам среды, а вы, в свою очередь, можете описать процесс развертывания в них своих контейнеров.

Azure Pipelines

Сервис Azure DevOps от Microsoft (ранее известный как Visual Studio Team Services) включает в себя механизм непрерывного развертывания под Azure Pipelines, аналогичный Google Cloud Build.

Компоненты непрерывного развертывания

Если у вас уже есть устоявшаяся система CD и вам просто нужно добавить компоненты для сборки контейнеров или их дальнейшего развертывания, мы предлагаем на выбор несколько вариантов, которые можно интегрировать в существующую инфраструктуру.

Docker Hub

Один из самых простых вариантов автоматической сборки контейнеров при изменении кода — использование Docker Hub (docs.docker.com/docker-hub/builds). Если у вас уже есть учетная запись в Docker Hub (см. раздел «Реестры контейнеров» на с. 59), вы можете создать триггер для репозитория в GitHub или BitBucket, который будет автоматически собирать и публиковать контейнеры в этом реестре.

Gitkube

Gitkube (gitkube.sh) — это инструмент для самостоятельного размещения, который работает в вашем кластере Kubernetes, наблюдает за Git-репозиторием и автоматически собирает и загружает новые контейнеры при срабатывании одного из ваших триггеров. Он очень простой, переносимый и легкий в настройке.

Flux

Инициирование процесса CD (или другой автоматизированной процедуры) в контексте веток или тегов Git иногда называют GitOps (www.weave.works/blog/gitops-operations-by-pull-request). Проект Flux (github.com/fluxcd/flux) является развитием этой идеи: он отслеживает изменения не в Git-репозитории, а в реестре контейнеров. Когда загружается новый контейнер, Flux автоматически развертывает его в вашем кластере Kubernetes.

Keel

Инструмент Keel (keel.sh), как и Flux, предназначен для развертывания новых образов контейнеров из реестра. Его можно сконфигурировать так, чтобы он реагировал на веб-хуки, отправлял и принимал сообщения через Slack, ждал одобрения перед развертыванием и выполнял другие полезные рабочие действия.

Процесс CD с использованием Cloud Build

Итак, вы познакомились с общими принципами CD и некоторыми доступными инструментами. Теперь рассмотрим полноценный пример конвейера CD от начала и до конца.

Вам вовсе не обязательно использовать именно те инструменты и конфигурацию, которые представлены здесь, — мы просто хотим, чтобы вы получили представление о том, как все это сочетается друг с другом, и могли адаптировать некоторые участки из примера к своей собственной среде.

В нашем примере мы будем использовать Google Cloud Platform (GCP), Google Kubernetes Engine clusters (GKE) и Google Cloud Build, но без каких-либо специфических функций этих продуктов. Вы можете воссоздать подобный процесс с помощью тех инструментов, которые вам больше нравятся.

Если вы хотите работать с этим примером с помощью своей собственной учетной записи GCP, имейте в виду, что здесь применяются платные услуги. Вряд ли вы разоритесь, но по завершении лучше удалить и почистить все облачные ресурсы, чтобы не переплачивать.

Настройка Google Cloud и GKE

Если вы впервые регистрируетесь в Google Cloud, вам полагается довольно существенный кредит, который должен позволить вам запускать кластеры Kubernetes и другие ресурсы на протяжении длительного периода времени и без каких-либо затрат. Узнать подробности и создать учетную запись можно на сайте Google Cloud Platform (cloud.google.com/free).

После того как вы зарегистрируетесь в Google Cloud и войдете в свой проект, следуйте инструкциям (cloud.google.com/kubernetes-engine/docs/how-to/creating-a-cluster), чтобы создать кластер GKE.

Затем инициализируйте Helm в своем кластере (см. раздел «Helm: диспетчер пакетов для Kubernetes» на с. 102).

Теперь, чтобы подготовить рабочий процесс, мы вместе пройдем через следующие этапы.

1. Создание копии репозитория `demo` в личной учетной записи GitHub.
2. Создание в Cloud Build триггера для сборки и тестирования в ответ на загрузку кода в любую ветку Git.
3. Создание триггера для развертывания в GKE на основе тегов Git.

Создание копии репозитория demo

Если у вас есть учетная запись GitHub, используйте веб-интерфейс, чтобы скопировать репозиторий `demo` (github.com/cloudnativedevops/demo).

Если вы не используете GitHub, создайте копию нашего репозитория и загрузите ее на свой собственный сервер Git.

Знакомство с Cloud Build

В Cloud Build, как и в Drone и во многих других современных платформах для CD, каждый этап процесса сборки состоит из запуска контейнера. Эти этапы описываются в виде файла YAML, который находится в вашем Git-репозитории.

Когда процесс инициируется в ответ на фиксацию изменений, Cloud Build создает копию репозитория на момент фиксации с определенной контрольной суммой (SHA) и выполняет каждый этап процесса по очереди.

Внутри репозитория `demo` находится папка с именем `hello-cloudbuild`. В ней вы найдете файл `cloudbuild.yaml`, который описывает конвейер Cloud Build.

Далее мы шаг за шагом пройдемся по каждому из этапов сборки.

Сборка контейнера с тестами

Вот первый шаг:

```
- id: build-test-image
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/docker
  entrypoint: bash
```

```
  args:
    - -c
    - |
      docker image build --target build --tag demo:test .
```

Как и все этапы в Cloud Build, этот состоит из пар «ключ — значение» в формате YAML:

- `id` — метка этапа сборки, понятная человеку;
- `dir` — определяет подкаталог Git-репозитория, в котором нужно работать;
- `name` — указывает на контейнер, который следует запустить на этом этапе;
- `entrypoint` — задает команду, которую нужно запустить в контейнере (вместо той, что указана по умолчанию);
- `args` — передает входной команде необходимые аргументы.

Вот и все!

Цель этого шага — собрать контейнер для выполнения тестов нашего приложения. Поскольку мы используем многоэтапную сборку (см. подраздел «Что собой представляет Dockerfile» на с. 56), сейчас займемся первым этапом и выполним следующую команду:

```
docker image build --target build --tag demo:test .
```

Аргумент `--target build` говорит Docker о том, что собрать нужно только ту часть Dockerfile, которая идет после `FROM golang:1.11-alpine AS build`, а затем следует остановиться, прежде чем переходить к следующему шагу.

Это означает, что в итоговом контейнере по-прежнему будет установлена среда Go и все пакеты или файлы, помеченные как `...AS build`. В сущности, мы создали временный контейнер, который используется лишь для тестирования нашего приложения и затем удаляется.

Выполнение тестов

Вот следующий этап:

```
- id: run-tests
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/docker
  entrypoint: bash
  args:
    - -c
    - |
      docker container run demo:test go test
```

Поскольку мы пометили наш временный контейнер как `demo:test`, его образ будет доступен на протяжении всего процесса сборки внутри Cloud Build. На этом этапе мы выполняем в нем команду `go test`. Если какой-то из тестов не будет пройден, сборка остановится и сообщит о неудаче. В противном случае мы перейдем к следующему шагу.

Собираем контейнер приложения

Здесь мы опять используем команду `docker build`, но уже без флага `--target`: это позволяет выполнить все этапы сборки и получить итоговый контейнер с приложением:

```
- id: build-app
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/docker
  entrypoint: bash
  args:
    - -c
    - |
      docker build --tag gcr.io/${PROJECT_ID}/demo:${COMMIT_SHA} .
```

Проверка манифестов Kubernetes

Итак, мы имеем контейнер, который прошел все тесты и готов к работе в Kubernetes. Но, чтобы его развернуть, мы используем чарт Helm. Таким образом, выполним команду `helm template`, чтобы сгенерировать манифесты Kubernetes, которые затем для проверки пропустим через инструмент `kubeval` (см. подраздел «kubeval» на с. 298):

```
- id: kubeval
  dir: hello-cloudbuild
  name: cloudnatived/helm-cloudbuilder
  entrypoint: bash
  args:
    - -c
    - |
      helm template ./k8s/demo/ | kubeval
```

Заметьте, здесь мы используем наш собственный образ контейнера Helm (cloudnatived/helm-cloudbuilder). Удивительно, но у инструмента, специально предназначенного для развертывания контейнеров, нет официального образа. Вы можете воспользоваться нашим, но в промышленной среде вам, скорее всего, захочется собрать свой собственный.

Публикация образа

Если процесс завершится успешно, Cloud Build автоматически опубликует итоговый образ контейнера в реестре. Чтобы указать, какие образы вы хотите опубликовать, перечислите их в разделе `images` файла Cloud Build:

```
images:
  - gcr.io/${PROJECT_ID}/demo:${COMMIT_SHA}
```

Теги на основе Git SHA

Что насчет тега `COMMIT_SHA`? В Git каждая фиксация кода имеет уникальный идентификатор под названием SHA (от англ. Secure Hash Algorithm — безопасный алгоритм хеширования, который его генерирует). Это длинная строка шестнадцатеричных цифр, такая как `5ba6bfd64a31eb4013ccaba27d95cddd15d50ba3`.

Пометив свой образ с помощью SHA, вы получите ссылку на фиксацию Git, из которой он сгенерирован и которая служит точным снимком кода, находящегося в контейнере. Использование исходных значений Git SHA в качестве тегов для элементов сборки позволяет собирать и тестировать большое количество функциональных веток одновременно без каких-либо конфликтов.

Итак, мы увидели, как работают конвейеры. Далее обратим наше внимание на триггеры, которые инициируют процесс сборки с соблюдением указанных нами условий.

Создание первого триггера сборки

Триггер Cloud Build определяет отслеживаемый Git-репозиторий, условие активации (такое как загрузка кода в определенную ветку или тег) и файл конвейера, который нужно выполнить.

Создайте новый триггер. Войдите в свой проект Google Cloud и перейдите по адресу https://console.cloud.google.com/cloud-build/triggers?pli=1.

Нажмите кнопку Add Trigger (Добавить триггер), чтобы создать новый триггер сборки, и выберите GitHub в качестве исходного репозитория.

Вас попросят выдать Google Cloud доступ к вашему репозиторию в GitHub: выберите `YOUR_GITHUB_USERNAME/demo`, и Google Cloud к нему подключится.

Затем сконфигурируйте триггер, как показано на рис. 14.1.

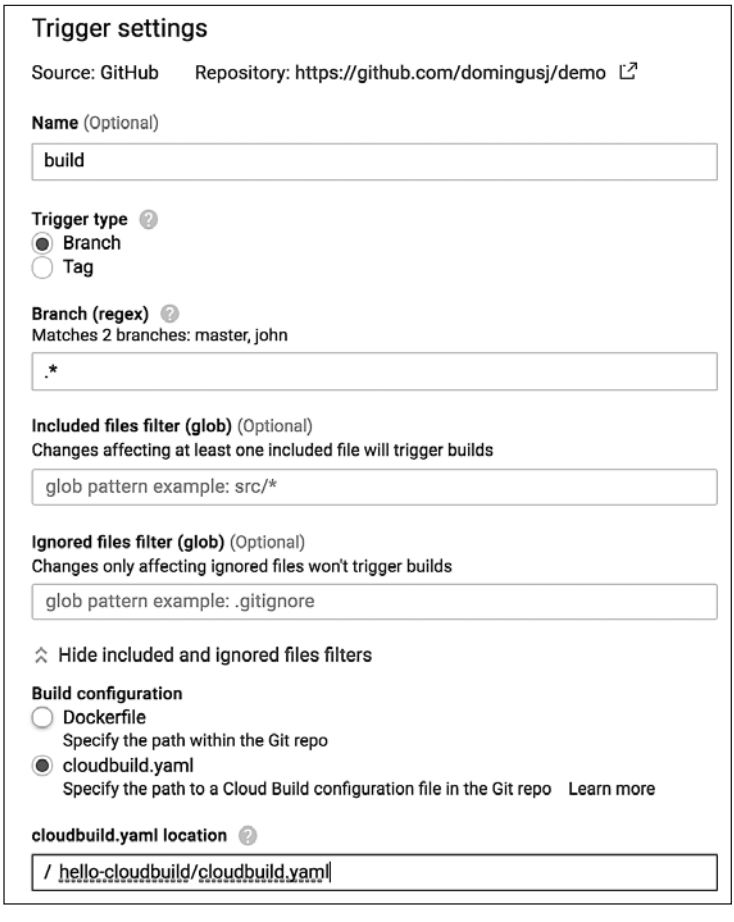

Рис. 14.1. Создание триггера

Триггер можно назвать так, как вам нравится. Оставьте в разделе `branch` значение по умолчанию `.*`, которое соответствует любой ветке.

Поменяйте раздел `Build configuration`, указав `cloudbuild.yaml` вместо `Dockerfile`.

Поле `cloudbuild.yaml Location` говорит Cloud Build, где следует искать наш файл конвейера с этапами сборки. В данном случае это `hello-cloudbuild/cloudbuild.yaml`.

Когда закончите, нажмите кнопку Create trigger (Создать триггер). Теперь все готово для того, чтобы мы могли проверить триггер и посмотреть, что произойдет!

Проверка триггера

Внесите какие-нибудь изменения в вашу копию репозитория `demo`. Например, создайте новую ветку и поменяйте приветствие с `Hello` на `Hola`:

```
cd hello-cloudbuild
git checkout -b hola
Switched to a new branch hola
```

Отредактируйте `main.go` и `main_test.go`, заменив `Hello` на `Hola` или на любое другое приветствие, которое вам нравится, и сохраните оба файла.

Выполните тесты и убедитесь в том, что все работает:

```
go test
PASS
ok   github.com/cloudnativedevops/demo/hello-cloudbuild 0.011s
```

Теперь зафиксируйте изменения и загрузите их в свою копию репозитория. Если все прошло хорошо, Cloud Build запустит новую сборку. Перейдите по адресу https://console.cloud.google.com/cloud-build/builds.

Вы увидите список последних сборок в вашем проекте. В самом его верху должна находиться сборка для изменения, которое вы только что загрузили. Она либо все еще работает, либо уже завершилась.

Надеемся, вы увидите зеленый флажок, который свидетельствует о прохождении всех этапов. В противном случае проверьте вывод сборки и посмотрите, что случилось.

Если все прошло хорошо, контейнер опубликуется в вашем закрытом реестре Google Container Registry, а его тег будет иметь контрольную сумму (SHA) фиксации вашего изменения в Git.

Развертывание из конвейера CD

Итак, вы можете инициировать сборку командой `git push`, запустить тесты и опубликовать итоговый контейнер в реестре. Теперь вы готовы развернуть этот контейнер в Kubernetes.

Давайте представим, что в данном примере у нас есть две среды — одна для промышленного использования (`production`), а другая — для заключительного тести-

рования (`staging`). Они будут развернуты в двух отдельных пространствах имен: `staging-demo` и `production-demo`.

Сконфигурируем Cloud Build так, чтобы изменения с тегами, содержащими `staging` и `production`, развертывались в соответствующей среде. Для этого нужно создать новый конвейер в отдельном файле YAML — `cloudbuild-deploy.yaml`. Далее описаны шаги, которые необходимо выполнить.

Получение учетных данных для кластера Kubernetes

Чтобы произвести развертывание в Kubernetes с использованием Helm, нужно сконфигурировать `kubectl` для общения с кластером:

```
- id: get-kube-config
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/kubectl
  env:
  - CLOUDSDK_CORE_PROJECT=${_CLOUDSDK_CORE_PROJECT}
  - CLOUDSDK_COMPUTE_ZONE=${_CLOUDSDK_COMPUTE_ZONE}
  - CLOUDSDK_CONTAINER_CLUSTER=${_CLOUDSDK_CONTAINER_CLUSTER}
  - KUBECONFIG=/workspace/.kube/config
  args:
    - cluster-info
```

Здесь мы ссылаемся на такие переменные, как `${_CLOUDSDK_CORE_PROJECT}`. Их можно определить либо в триггере сборки (это будет сделано в нашем примере), либо в самом конвейере в разделе `substitutions`:

```
substitutions:
  _CLOUDSDK_CORE_PROJECT=demo_project
```

Пользовательские подстановки должны начинаться с подчеркивания (`_`) и содержать в имени только числа и прописные буквы. Кроме того, Cloud Build предоставляет некоторые готовые подстановки, такие как `$PROJECT_ID` и `$COMMIT_SHA`. Полный список находится здесь: cloud.google.com/cloud-build/docs/configuring-builds/substitute-variable-values.

Вы также должны выдать служебной учетной записи Cloud Build права на изменение вашего кластера Kubernetes Engine. В разделе IAM консоли GCP назначьте служебной учетной записи Cloud Build своего проекта IAM-роль *Kubernetes Engine Developer*.

Добавление тега environment

На этом этапе мы пометим контейнер тем же тегом Git, который инициировал развертывание:

```
- id: update-deploy-tag
  dir: hello-cloudbuild
  name: gcr.io/cloud-builders/gcloud
  args:
    - container
    - images
    - add-tag
    - gcr.io/${PROJECT_ID}/demo:${COMMIT_SHA}
    - gcr.io/${PROJECT_ID}/demo:${TAG_NAME}
```

Развертывание в кластере

Здесь мы запускаем Helm, чтобы обновить само приложение в кластере, и используем полученные ранее учетные данные Kubernetes:

```
- id: deploy
  dir: hello-cloudbuild
  name: cloudnatived/helm-cloudbuilder
  env:
    - KUBECONFIG=/workspace/.kube/config
  args:
    - helm
    - upgrade
    - --install
    - ${TAG_NAME}-demo
    - --namespace=${TAG_NAME}-demo
    - --values
    - k8s/demo/${TAG_NAME}-values.yaml
    - --set
    - container.image=gcr.io/${PROJECT_ID}/demo
    - --set
    - container.tag=${COMMIT_SHA}
    - ./k8s/demo
```

Мы передаем команде `helm upgrade` несколько дополнительных флагов:

- `namespace` — пространство имен, в котором должно быть развернуто приложение;
- `values` — файл `values.yaml` для Helm, который будет использоваться в этой среде;
- `set container.image` — задает имя развертываемого контейнера;
- `set container.tag` — развертывает образ с этим конкретным тегом (оригинальной контрольной суммой Git SHA).

Создание триггера развертывания

Теперь добавим триггеры для развертывания в `staging` и `production`.

Создайте в Cloud Build новый триггер, как вы это делали в подразделе «Создание первого триггера сборки» на с. 320, но на этот раз сконфигурируйте его так, чтобы он срабатывал при загрузке тега, а не ветки.

Вместо файла `hello-cloudbuild/cloudbuild.yaml` в этой сборке будет использоваться `hello-cloudbuild/cloudbuild-deploy.yaml`.

Укажите в разделе `Substitution variables` переменные, предназначенные для сборки `staging`:

- переменной `_CLOUDSDK_CORE_PROJECT` нужно присвоить идентификатор вашего проекта Google Cloud, в котором выполняется ваш кластер GKE;
- переменная `_CLOUDSDK_COMPUTE_ZONE` должна совпадать с зоной доступности вашего кластера (или регионом, если кластер региональный);
- `_CLOUDSDK_CONTAINER_CLUSTER` — это имя вашего кластера GKE.

Благодаря этим переменным можно использовать один и тот же файл YAML для развертывания в `staging` и `production`, даже если необходимо разместить эти среды в разных кластерах или отдельных проектах GCP.

Чтобы проверить свой новый триггер `staging`, загрузите в репозиторий одноименный тег:

```
git tag -f staging
git push -f origin refs/tags/staging
Total 0 (delta 0), reused 0 (delta 0)
To github.com:domingusj/demo.git
* [new tag]          staging -> staging
```

Как и прежде, вы можете наблюдать за процессом сборки (console.cloud.google.com/projectselector/cloud-build/builds?supportedpurview=project&pli=1).

Если все пройдет согласно плану, Cloud Build успешно аутентифицируется в вашем кластере GKE и развернет в пространстве имен `staging-demo` финальную тестовую версию приложения.

Это можно проверить на панели GKE (или с помощью `helm status`).

Наконец, выполните те же шаги для создания триггера, который в ответ на загрузку тега `production` развернет приложение в промышленной среде.

Оптимизация процесса сборки

Если вы используете средство для CD на основе контейнеров, такое как Cloud Build, важно минимизировать размер контейнера на каждом этапе (см. подраздел «Минимальные образы контейнеров» на с. 56). Ведь, когда сборка выполняется десятки и сотни раз в день, увеличенное время ожидания загрузки громоздких контейнеров дает о себе знать.

Например, если вы расшифровываете конфиденциальные данные с помощью Sops (см. раздел «Шифрование конфиденциальных данных с помощью Sops» на с. 252), официальный образ контейнера `mozilla/sops` будет занимать примерно 800 МиБ. Но, собрав свой собственный образ в ходе многоэтапной сборки, вы сможете уменьшить его размер где-то до 20 МиБ. Поскольку образ загружается во время каждой сборки, сокращение его объема в 40 раз того стоит.

Вы можете загрузить (github.com/bitfield/sops) версию Sops с модифицированным файлом Dockerfile для построения минимального образа контейнера.

Адаптация демонстрационного конвейера

Мы надеемся, что на этом примере нам удалось продемонстрировать ключевые концепции процесса CD. Если вы используете Cloud Build, данный код может стать отправной точкой для построения вашего собственного конвейера. Если же вы применяете другие инструменты, адаптация представленных здесь шагов для работы в вашей собственной среде должна быть относительно простой.

Резюме

Подготовка процесса непрерывного развертывания приложений позволит вам доставлять свое ПО предсказуемо, надежно и быстро. В идеале при загрузке кода в репозиторий системы контроля версий все этапы сборки, тестирования и развертывания должны происходить автоматически в централизованном конвейере.

Поскольку в сфере CD существует богатый выбор решений и методик, мы не можем предложить вам универсальный рецепт, подходящий для всех. Вместо этого мы попытались показать, за счет чего и почему процесс CD является полезным, и выделили несколько моментов, над которыми стоит задуматься при его реализации в своей компании.

- При построении нового конвейера очень важен выбор инструмента для CD. Все решения, которые мы упоминаем в этой книге, с большой долей вероятности можно интегрировать с почти любым инструментом для CD.

- Jenkins, GitLab, Drone, Cloud Build и Spinnaker — это лишь часть популярных инструментов для CD, хорошо совместимых с Kubernetes. Существует множество более новых решений, таких как Gitkube, Flux и Keel, специально созданных для автоматизации развертываний в кластерах Kubernetes.
- Определение этапов процесса сборки в виде кода позволяет отслеживать и модифицировать их вместе с исходным кодом приложения.
- Контейнеры позволяют разработчикам прогонять результаты сборки через разные среды: от тестовой до среды финального тестирования и промышленной. В идеале для этого не нужно собирать новые контейнеры.
- Наш демонстрационный конвейер на основе Cloud Build должен легко адаптироваться к другим инструментам и типам приложений. В целом этапы сборки, тестирования и развертывания являются одинаковыми в любом процессе CD, независимо от инструментов или типа программного обеспечения.

15 Наблюдаемость и мониторинг

> На борту корабля никогда ничего не идет в точности так, как нужно.
>
> *Уильям Лангуиш. The Outlaw Sea*

В этой главе мы рассмотрим вопрос наблюдаемости и мониторинга для облачно-ориентированных приложений. Что такое наблюдаемость? Как она относится к мониторингу? Как выполнять мониторинг и трассировку и как вести журнал в Kubernetes?

Что такое наблюдаемость

Термин *«наблюдаемость»* вам может быть не знаком, но его все чаще используют для обозначения более обширной, чем мониторинг, области. Сначала мы обсудим *мониторинг*, а затем посмотрим, как наблюдаемость его расширяет.

Что такое мониторинг

Все ли хорошо с вашим сайтом? Проверьте, мы подождем. Самый простой способ убедиться в том, что все ваши приложения и сервисы работают как следует, — посмотреть на них собственными глазами. Но когда речь идет о мониторинге в контексте DevOps, в основном имеется в виду *автоматизированный мониторинг*.

Автоматизированный мониторинг — это проверка доступности и поведения сайта или сервиса каким-то программным способом, обычно по расписанию и с поддержкой автоматического оповещения инженеров в случае возникновения проблемы. Но как определить проблему?

Мониторинг методом черного ящика

Сейчас мы рассмотрим простой пример со статическим сайтом, таким как блог cloudnativedevopsblog.com, сопровождающий эту книгу.

Если сайт вообще не работает, он либо не ответит, либо вернет браузеру сообщение об ошибке (надеемся, что это не так, но ничто не идеально). Поэтому самым простым методом мониторинга сайта будет загрузка его домашней страницы и проверка кода ответа (200 свидетельствует об успешном запросе). Делать это следует с помощью консольного HTTP-клиента, такого как `httpie` или `curl`. Если клиент вернул ненулевой код возврата, это говорит о проблеме с загрузкой сайта.

Но представьте, что проблема закралась в конфигурацию веб-сервера, и, несмотря на HTTP-ответ 200 OK, вам возвращается пустая страница (или какое-то стандартное приветствие, или, может быть, вообще не тот сайт). Наша простая проверка этого не заметит, так как HTTP-запрос завершится успешно, но сам сайт по-прежнему будет недоступен для пользователей и они не смогут читать наши удивительные и информативные статьи.

Более продуманная проверка могла бы искать на странице определенный текст, такой как *Cloude Native DevOps*. Это позволило бы обнаружить проблему с неправильно сконфигурированным, но рабочим веб-сервером.

Не статическими страницами едиными

Легко можно догадаться, что динамическим сайтам требуется более сложный мониторинг. Например, если на сайте есть механизм для входа в систему, средство мониторинга может попытаться им воспользоваться, предоставив заранее подготовленную учетную запись, и отправить уведомление в случае неудачи. Или если на сайте есть поисковая строка, в нее можно ввести какой-то текст, симулировать нажатие кнопки поиска и проверить наличие каких-то данных в результатах.

В случае с простым сайтом для ответа на вопрос «Работает ли он?» может быть достаточно простого *«да/нет»*. Но для облачно-ориентированных приложений, которые обычно представляют собой более сложные распределенные системы, вопрос может состоять из нескольких частей.

- Доступно ли мое приложение в любой точке планеты или только в некоторых регионах?
- Сколько времени уходит на его загрузку у большинства моих пользователей?
- А как обстоят дела у пользователей с низкой скоростью загрузки?
- Все ли функции моего сайта работают так, как было задумано?

- Не слишком ли медленно работают некоторые функции и сколько пользователей от этого страдает?
- Что случится с моим приложением, если сторонний сервис, на который оно полагается, недоступен или неисправен?
- Что произойдет, если у моего облачного провайдера случится перебой в работе?

Становится ясно, что в мире мониторинга облачно-ориентированных распределенных систем мало что ясно.

Ограничения мониторинга методом черного ящика

Какими бы сложными ни были эти проверки, все они относятся к одной категории: *мониторинг методом черного ящика*. Как можно догадаться из названия, такой мониторинг наблюдает только за внешним поведением системы, не пытаясь выяснить, что происходит внутри.

Еще несколько лет назад использование мониторинга методом черного ящика в исполнении таких популярных инструментов, как Nagios, Icinga, Zabbix, Sensu и Check_MK, считалось передовым подходом. Безусловно, *любые* автоматизированные проверки системы, по сравнению с их отсутствием, являются большим шагом вперед. Но у подобных проверок есть свои недостатки.

- Они могут обнаружить только предсказуемые сбои (например, сайт перестал отвечать).
- Их можно применять только к тем частям системы, которые доступны извне.
- Они являются пассивными и реактивными: сообщают о проблеме только *после* того, как что-то случилось.
- Они могут ответить на вопрос «Что сломалось?», но не могут объяснить почему, а это более важно.

Чтобы ответить на вопрос «Почему?», мы должны выйти за рамки мониторинга методом черного ящика.

У проверки вида *«работает/не работает»* есть еще одна проблема: а что мы вообще понимаем под «работает»?

Что означает «работает»

В системном администрировании устойчивость и доступность приложений привыкли измерять *временем доступности* и выражать это в процентах. Например, приложение с временем доступности 99 % бывает недоступно не более чем 1 % от

заданного периода времени. Время доступности 99,9 % называют *тремя девятками*: это значит, что система простаивает примерно девять часов в год, что считается хорошим показателем для среднего веб-приложения. Четыре девятки (99,99 %) допускают время простоя не больше часа в год, а пять девяток (99,999 %) — не больше пяти минут.

В результате можно подумать, что чем больше девяток, тем лучше. Но смотреть на вещи таким образом — значит упускать кое-что важное.

> Девятки неважны, если пользователи недовольны.
>
> *Чарити Мэйджорс*[1]

Девятки неважны, если пользователи недовольны

Как говорится, что можно измерить, то можно и приукрасить. Поэтому очень осторожно подходите к тому, что замеряете. Если для пользователей сервис не работает, ваши внутренние показатели не имеют значения — *сервис недоступен*. Приложение может не удовлетворять пользователей и по множеству других причин, даже если формально оно *работает*.

Возьмем очевидный пример: что, если на загрузку вашего сайта уходит десять секунд? Загрузившись, он будет нормально работать, но столь долгое время ответа делает его бесполезным — пользователи просто уйдут в другое место.

Традиционные системы мониторинга методом черного ящика могут попытаться решить проблему, определив допустимое время ожидания пять секунд: в случае превышения данного значения будет сгенерировано оповещение, а сервис станет считаться нерабочим. А если время загрузки колеблется от двух до десяти секунд? С таким жестким лимитом сервис может оказаться *рабочим* для одних пользователей и *нерабочим* для других. Что, если у пользователей из Северной Америки все хорошо, а посетители из Европы и Азии сталкиваются с недопустимым временем ожидания?

Облачно-ориентированные приложения никогда не являются рабочими

Вы можете придумать более сложные правила и лимиты, чтобы ответ о состоянии сервиса всегда был однозначным, но дело в том, что сам вопрос непоправимо ошибочен. Распределенные системы, такие как облачно-ориентированные приложения,

[1] red.ht/2FMZcMZ.

никогда нельзя считать рабочими (red.ht/2hMHwSL): они постоянно находятся в состоянии частично ограниченного уровня обслуживания.

Мы показали пример целого класса проблем под названием *«серые сбои»* (blog.acolyer.org/2017/06/15/gray-failure-the-achilles-heel-of-cloud-scale-systems). Их по определению сложно обнаружить — особенно если смотреть только под одним углом или если провести единичное наблюдение.

Таким образом, мониторинг методом черного ящика является хорошим первым шагом на пути к достижению наблюдаемости, но важно понимать, что на этом нельзя останавливаться. Стоит поискать лучший метод.

Ведение журнала

Большинство приложений генерируют разного рода *журнальные записи*, как правило, с временными метками, которые указывают на то, когда и в каком порядке записи были сохранены. Например, веб-сервер записывает в свой журнал такую информацию о каждом запросе:

- запрошенный URI-адрес;
- IP-адрес клиента;
- HTTP-статус ответа.

Столкнувшись с ошибкой, приложение обычно записывает этот факт в журнал и добавляет сведения, которые могут помочь системным администраторам разобраться в причинах появления ошибки.

Часто журнальные записи, принадлежащие широкому спектру приложений и сервисов, *агрегируются* в центральную базу данных (например, Elasticsearch). Там их можно искать и визуализировать для более простой отладки. Такие инструменты, как Logstash и Kibana, или самостоятельно размещаемые сервисы наподобие Splunk и Loggly созданы для того, чтобы помочь вам собирать и анализировать большие объемы журнальных данных.

Ограничения журнальных записей

Журнальные записи могут быть полезными, но и у них есть ограничения. Решение о том, что следует записывать, а что — нет, принимается программистом на этапе написания приложения. Таким образом, как и проверки методом черного ящика, журнальные записи отвечают только на те вопросы и обнаруживают только те проблемы, которые можно предсказать заранее.

Кроме того, извлечение информации из журналов может оказаться непростой задачей, так как каждое приложение ведет их в своем формате. Поэтому администраторам часто приходится писать собственные анализаторы для каждого типа журнальных записей, чтобы превратить их в пригодные к использованию числовые или событийные данные.

Поскольку информации, которая записывается в журнал, должно быть достаточно для диагностики всевозможного рода проблем, обычно она имеет довольно высокий уровень шума. Если записывать все подряд, для поиска нужного вам сообщения об ошибке придется перелопатить сотни страниц и потратить много времени. Если же в журнал попадают только ошибки, вам будет сложно понять, как выглядит *нормальная* работа.

Журнальные записи тяжело масштабировать

Журнальные записи не очень хорошо масштабируются в соответствии с изменением трафика. Если каждый пользовательский запрос будет генерировать строчку в журнале, которую надо отправить агрегатору, на это может уйти большая часть пропускной способности вашей сети. Сеть будет недоступной для обслуживания пользователей, а ваш агрегатор станет слабым звеном в производительности системы.

Многие журнальные сервисы взимают плату за объем записей, которые вы генерируете, — это логично, но не очень приятно. Однако у вас появляется финансовый стимул записывать меньше информации, иметь меньше пользователей и обслуживать меньше трафика!

То же самое относится и к журнальным решениям с самостоятельным размещением: чем больше данных вы храните, тем больше вы платите за оборудование, место на диске и сетевые ресурсы и тем больше времени приходится тратить инженерам, чтобы поддерживать агрегацию журналов в рабочем состоянии.

Имеет ли смысл ведение журнала в Kubernetes?

Мы уже упоминали о том, как контейнеры генерируют журнальные записи и как их можно исследовать непосредственно в Kubernetes (см. подраздел «Просмотр журнальных записей контейнера» на с. 167). Данная методика отладки хорошо подходит для отдельных контейнеров.

Если вы все же ведете журнал, то вместо записей в виде обычного текста вам следует использовать какой-то формат структурированных данных наподобие JSON,

который можно анализировать автоматически (см. раздел «Процесс наблюдаемости» на с. 338).

Централизованная агрегация журналов (с сохранением их в таких сервисах, как ELK) может быть полезна в приложениях Kubernetes, но одной ее недостаточно. Несмотря на то что данный подход имеет несколько бизнес-применений (например, таковы требования к аудиту и безопасности или это нужно для анализа информации о клиентах), журналы не могут дать вам всю информацию, необходимую для обеспечения настоящей наблюдаемости.

Для этого нужно пойти дальше и обратиться к чему-то значительно более мощному.

Введение в показатели

Более сложный способ сбора информации о сервисах заключается в использовании *показателей*. Как понятно из названия, показатель — это числовая мера чего-либо. В зависимости от приложения вам могут пригодиться следующие показатели:

- количество запросов, которые обрабатываются в данный момент;
- количество запросов, обрабатываемых в минуту (секунду, час);
- количество ошибок, возникающих при обработке запросов;
- среднее время обслуживания запросов (или пиковое время, или 99-й перцентиль).

Полезно иметь сведения не только о приложениях, но и о таких аспектах работы вашей инфраструктуры, как:

- процессорные ресурсы, занятые отдельными процессами или контейнерами;
- активность дискового ввода/вывода на узлах и серверах;
- входящий и исходящий сетевой трафик компьютеров, кластеров или балансировщиков нагрузки.

Показатели помогают понять причину проблемы

Показатели выводят мониторинг на новый уровень, за пределы логики вида *«работает/не работает»*. По аналогии со спидометром в машине или температурной шкалой в термометре они дают числовую информацию о том, что происходит. В отличие от журнальных записей, показатели легко обрабатываются разными способами: можно рисовать графики, формировать статистику или оповещать о достижении заранее заданных лимитов. Например, ваша система мониторинга способна уведомить о том, что в заданный период времени частота возникновения ошибок превысила 10 %.

Показатели также могут помочь с поиском *причин* возникновения проблем. Представьте ситуацию, когда пользователи начали получать ответ от вашего приложения с высокой задержкой *(латентностью)*. Вы проверяете и видите, что скачок соответствующего показателя совпадает с аналогичным скачком в использовании *процессора* для определенного компьютера или компонента. И это сразу же дает вам подсказку, где следует искать проблему. Возможно, компонент затрудняет работу кластера или многократно пытается выполнить какую-то неудачную операцию, а может быть, его узел столкнулся с аппаратными проблемами.

Показатели помогают предсказать проблемы

Показатели бывают и *прогностическими*: обычно проблемы не возникают в один миг. Прежде чем проблема станет заметной вам или вашим пользователям, на ее скорое появление может указать всплеск какого-либо показателя.

Например, показатель использования диска для сервера может медленно расти и в какой-то момент достичь уровня, когда на диске не останется свободного места и система начнет отказывать. Если вы будете оповещены об этом показателе до того, как он достигнет критического значения, вы успеете предотвратить сбой.

Некоторые системы даже используют методы машинного обучения для анализа показателей, обнаружения аномалий и поиска причин сбоев. Особенно это может быть полезно в сложных распределенных системах, но в большинстве случаев вполне достаточно сбора и визуализации показателей в сочетании с оповещениями.

Показатели собираются внутри приложений

При проверке методом черного ящика администраторы должны иметь представление о внутреннем устройстве приложения или сервиса, чтобы предсказывать возможные виды проблем и то, как те могут повлиять на внешнее поведение. Для сравнения: показатели дают возможность разработчикам приложения извлекать информацию о скрытых аспектах системы, основываясь на своем понимании того, как она на самом деле работает (и отказывает).

> Перестаньте заниматься реверс-инжинирингом приложений и начните мониторить их изнутри.
>
> *Келси Хайтауэр[1]. Monitorama 2016*

[1] vimeo.com/173610242.

Для сбора и администрирования показателей широко используются такие инструменты, как Prometheus, statsd и Graphite, а также управляемые сервисы, включая Datadog, New Relic и Dynatrace.

В главе 16 мы поговорим о показателях намного более подробно и среди прочего попытаемся объяснить, какие из них заслуживают вашего особого внимания и что с ними нужно делать. А пока закончим наш обзор наблюдаемости разговором о трассировке.

Трассировка

Еще одной полезной методикой мониторинга является *трассировка*. Особенно большое значение она имеет в распределенных системах. Если показатели и журнальные записи говорят о том, что происходит с тем или иным компонентом кластера, трассировка отслеживает отдельно взятый пользовательский запрос на протяжении всего его жизненного цикла.

Допустим, вы пытаетесь понять, почему запросы некоторых пользователей демонстрируют очень высокую задержку выполнения. Вы проверяете показатели каждого системного компонента: балансировщика нагрузки, условного контроля доступа, веб-сервера, сервера приложений, базы данных, шины сообщений и т. д. Но в итоге все выглядит нормально. Так что же происходит?

Если трассировать отдельный (желательно репрезентативный) запрос от момента открытия пользовательского соединения до момента его закрытия, вы получите картину того, из чего состоит задержка выполнения на каждом этапе путешествия запроса по системе.

Например, вы можете обнаружить, что время, потраченное на обработку запроса, находится в пределах нормы на каждом из этапов этого процесса, за исключением прохождения через базу данных — там его затрачивается в 100 раз больше, чем обычно. Сама база данных работает нормально, и показатели не сигнализируют о проблемах, но по какой-то причине серверу приложений приходится ждать ее ответа очень долгое время.

В конце концов вы установили, что проблема связана с чрезмерной потерей пакетов на одном конкретном сетевом узле, расположенном между серверами приложений и сервером базы данных. Не имей вы возможности *понаблюдать за запросом* с помощью распределенной трассировки, вам было бы сложно найти подобного рода проблему.

В число популярных инструментов для распределенной трассировки входят Zipkin, Jaeger и LightStep. Инженер Масрур Хасан написал полезную статью

(medium.com/@masroor.hasan/tracing-infrastructure-with-jaeger-on-kubernetes-6800132a677) о том, как использовать Jaeger для распределенной трассировки в Kubernetes.

Фреймворк OpenTracing (opentracing.io) (входящий в Cloud Native Computing Foundation) стремится предоставить стандартный набор API и библиотек для распределенной трассировки.

Наблюдаемость

Поскольку термин *«мониторинг»* может иметь разное значение для разных людей — от старых добрых проверок методом черного ящика до сочетания показателей, ведения журнала и трассировки — все эти методики в совокупности все чаще называют *наблюдаемостью*. Наблюдаемость — это совокупность методов и процессов, помогающих узнать, насколько хорошо система оснащена и насколько просто можно определить, что происходит внутри нее. Некоторые люди считают, что наблюдаемость включает в себя классический мониторинг, но существует и мнение, что она отражает совершенно отдельный образ мышления.

Наверное, наиболее доходчиво разделить эти термины можно так: мониторинг определяет, *работает ли система*, тогда как наблюдаемость предлагает задаться вопросом, *почему она не работает*.

Наблюдаемость основана на понимании

В более общем смысле наблюдаемость заключается в *понимании*: понимании того, что делает ваша система и как она это делает. Например, если выкатить изменение, призванное улучшить производительность конкретной функции на 10 %, то наблюдаемость покажет, удалось ли этого достичь. Если производительность выросла незначительно или, что еще хуже, немного снизилась, необходимо пересматривать код.

В то же время, если производительность выросла на 20 %, изменение превзошло ваши ожидания и, возможно, вам нужно подумать, почему прогнозы не оправдались.

Наблюдаемость помогает сформировать и улучшить понимание того, как взаимодействуют разные части вашей системы.

Наблюдаемость также имеет прямое отношение к *данным*. Вам следует знать, какие данные генерировать, что собирать, как это агрегировать (если есть необходимость), на каких результатах нужно сосредоточиться и как их запрашивать/отображать.

Программное обеспечение является непрозрачным

В традиционном мониторинге мы имеем много данных о *внутренних механизмах*: нагрузке на процессор, дисковой активности, сетевых пакетах и т. д. Но, опираясь на них, сложно рассуждать, что делает *программное обеспечение*. Для этого само ПО должно поддаваться измерению и оценке.

> Программное обеспечение изначально является непрозрачным. Чтобы люди могли понимать, чем оно занимается, оно должно генерировать данные. Наблюдаемые системы позволяют судить о том, работают ли ПО как следует, и, если нет, дают возможность определить масштаб проблемы и выяснить, что пошло не так.
>
> *Кристин Спанг (Nylas)*[1]

Формирование культуры наблюдаемости

В еще более общем смысле наблюдаемость основана на *культуре*. Это ключевой принцип DevOps, который замыкает цикл между разработкой кода и его выполнением в промышленном масштабе. Наблюдаемость — основной инструмент для завершения этого цикла. Разработчики и системные администраторы должны тесно сотрудничать для обеспечения наблюдаемости, им стоит выработать наилучший способ использования собранной информации и реакции на нее.

> Цель команды, которая занимается наблюдаемостью, не в том, чтобы собирать журнальные записи, показатели и трассировки. Такая команда должна сформировать культуру проектирования, основанную на фактах и обратной связи, и затем распространить эту культуру по всей организации.
>
> *Брайан Нокс (DigitalOcean)*

Процесс наблюдаемости

Как работает наблюдаемость с практической точки зрения? Довольно часто мы имеем дело с несколькими источниками данных (журнальными записями, показателями и т. д.), которые по необходимости подключаются к различным хранилищам.

---

[1] twitter.com/jetarrant/status/1025122034735435776.

Например, ваши журнальные записи могут отправляться на сервер ELK, показатели могут распределяться между тремя или четырьмя разными управляемыми сервисами, а для отчетов традиционного мониторинга может быть предусмотрен еще один сервис. Такая ситуация далека от идеала.

Во-первых, это сложно масштабировать. Чем больше источников и хранилищ данных у вас есть, тем теснее они между собой связаны и тем больше трафика проходит между ними. Нет особого смысла направлять усилия инженеров на то, чтобы поддерживать все эти разные связи в стабильном и надежном состоянии.

К тому же чем теснее ваша система интегрирована с определенными решениями или поставщиками, тем сложнее их поменять и попробовать альтернативные варианты.

Все более популярным способом решения этой проблемы становится *процесс наблюдаемости* (dzone.com/articles/the-observability-pipeline).

> Налаживая процесс наблюдаемости, мы отделяем источники данных от их потребителей и предоставляем буфер. Это делает данные легкопотребляемыми. Больше не нужно думать над тем, какую информацию следует отправлять из контейнеров, виртуальной машины и инфраструктуры, куда она должна поступать и как мы ее будем передавать. Вместо этого все данные направляются в конвейер, который занимается их фильтрацией и распределением по подходящим местам. Это также дает нам большую гибкость в отношении добавления/удаления исходящих каналов и создает буфер между производителями и потребителями информации.
>
> *Тайлер Трет*

Процесс наблюдаемости имеет огромные преимущества. Благодаря ему источник данных можно добавить, просто подключив к вашему конвейеру. Аналогично новый сервис для визуализации или оповещений становится с точки зрения конвейера лишь еще одним потребителем.

Поскольку данные в конвейере буферизируются, ничего никогда не теряется. При внезапном скачке трафика и слишком большом количестве показателей конвейер не станет отклонять эту информацию, а просто поместит ее в буфер.

Применение процесса наблюдаемости требует стандартного формата показателей (см. подраздел «Prometheus» на с. 371) и в идеале структурированного ведения журналов приложений с помощью JSON или любого другого формата сериализованных данных. Вместо того чтобы генерировать журнальные записи в виде обычного текста, который затем нужно будет анализировать с помощью хрупких регулярных выражений, лучше с самого начала позаботиться о структурировании своих данных.

Мониторинг в Kubernetes

Теперь мы немного лучше понимаем, что такое мониторинг методом черного ящика и как он в целом соотносится с наблюдаемостью. Посмотрим, как все это применить к приложениям Kubernetes.

Внешние проверки методом черного ящика

Как мы уже видели, мониторинг методом черного ящика определяет только то, работает ли ваше приложение (хотя это тоже очень полезная информация). Несмотря на всевозможные неполадки, облачно-ориентированное приложение может продолжать обслуживать некоторые запросы на вполне приемлемом уровне: инженеры будут работать над устранением внутренних проблем, таких как медленные запросы к базе данных и повышенная частота возникновения ошибок, а пользователи об этом даже не догадаются.

Однако в случае возникновения проблем более серьезного рода произойдет полномасштабный *сбой*: приложение станет недоступным или перестанет работать для большинства пользователей. Это может быть плохо не только для ваших клиентов, но и для вашего бизнеса (в зависимости от приложения). Чтобы обнаружить такой сбой, система мониторинга должна взаимодействовать с сервисами так, как это делает обычный пользователь.

Мониторинг симулирует поведение пользователя

Например, если это HTTP-сервис, система мониторинга должна обращаться к нему с помощью HTTP-запросов, а не TCP-соединений. Если сервис всего лишь возвращает статический текст, можно проверить, содержится ли в нем какая-либо строка, которая там должна быть. Обычно все немного сложнее, и, как мы видели в подразделе «Мониторинг методом черного ящика» на с. 329, ваши проверки тоже могут быть более изощренными.

Однако в ситуации сбоя в работе вполне возможно, что простого сравнения текста окажется достаточно для того, чтобы обнаружить отказ вашего приложения. Но выполнение таких проверок методом черного ящика изнутри вашей инфраструктуры (например, в Kubernetes) — не самая лучшая идея. Сбой может быть вызван всевозможными проблемами между пользователем и внешней границей вашей инфраструктуры, включая такие:

- неправильные DNS-записи;
- разделение сети;

- потеря пакетов;
- плохо сконфигурированные маршрутизаторы;
- некачественные правила брандмауэра или их отсутствие;
- перебои в работе облачного провайдера.

В таких ситуациях механизмы внутренних показателей и мониторинга могут не обнаружить никаких проблем. Поэтому основной задачей с точки зрения наблюдаемости должно быть отслеживание доступности ваших сервисов из какой-то точки, размещенной за пределами вашей инфраструктуры. Такой мониторинг умеют выполнять многочисленные сторонние сервисы (их иногда называют *«мониторинг как услуга»* (monitoring as a service, MaaS)), включая Uptime Robot, Pingdom и Wormly.

Не нужно строить свою собственную инфраструктуру мониторинга

У большинства этих сервисов есть либо бесплатный тариф, либо довольно дешевая подписка. Но сколько бы вы за них ни платили, это следует относить к основным операционным расходам. Не пытайтесь построить свою собственную инфраструктуру для внешнего мониторинга — оно того не стоит. Скорее всего, годовая профессиональная подписка на Uptime Robot обойдется вам дешевле, чем один час работы вашего инженера.

При выборе провайдера внешнего мониторинга обращайте внимание на следующие важнейшие возможности.

- Проверки по HTTP/HTTPS.
- Обнаружение некорректности или истечения срока годности вашего TLS-сертификата.
- Сопоставление ключевых слов (оповещения об отсутствии *или* наличии ключевого слова).
- Автоматическое создание или обновление проверок через API.
- Оповещение по электронной почте, через SMS, веб-хуки... Или какой-то другой простой механизм.

На страницах этой книги мы продвигаем идею *инфраструктуры как кода*, поэтому у вас также должна быть возможность программной автоматизации вашего внешнего мониторинга. Например, Uptime Robot предлагает простой интерфейс REST API для создания новых проверок. Вы можете автоматизировать его с помощью клиентской библиотеки или утилиты командной строки наподобие `uptimerobot` (github.com/bitfield/uptimerobot).

Не так уж важно, какой именно сервис внешнего мониторинга вы используете, — просто не забывайте об этом аспекте. Но не нужно на этом останавливаться. В следующем разделе мы покажем, каким образом можно отслеживать работоспособность приложений внутри самого кластера Kubernetes.

Внутренние проверки работоспособности

Сбои, происходящие в облачно-ориентированных приложениях, бывают сложными, непредсказуемыми и незаметными. Приложения должны устойчиво работать и уметь ограничивать свою производительность в случае непредвиденных проблем. Но, как ни странно, чем более устойчивыми они являются, тем сложнее обнаруживать такие сбои методом черного ящика.

Чтобы решить эту проблему, приложению следует выполнять свои собственные проверки работоспособности. Разработчик конкретной функции или сервиса лучше всего знает, что именно должно оставаться *работоспособным*. Поэтому он или она может написать код таким образом, чтобы выдаваемые результаты можно было отслеживать из внешнего контейнера (например, в виде конечной точки, доступной по HTTP).

Довольны ли пользователи?

Как мы уже видели в подразделе «Проверки работоспособности» на с. 111, Kubernetes предоставляет приложениям простой механизм, с помощью которого они могут сигнализировать о своей работоспособности или готовности: это хорошая отправная точка. Обычно такие проверки выглядят довольно просто — приложение всегда отвечает «OK» на любой запрос. Если ответа нет, Kubernetes считает приложение нерабочим или неготовым.

Но, в чем убедились многие программисты на своем горьком опыте, сам факт выполнения программы вовсе не означает, что она работает корректно. Более сложная проверка готовности должна отвечать на вопрос «Что этому приложению нужно для работы?».

Например, если оно должно общаться с базой данных, вы можете проверить наличие корректного и отзывчивого соединения с сервером базы. Если зависит от других сервисов, вы можете проверить их доступность. Поскольку проверки работоспособности проводятся часто, они не должны быть слишком ресурсоемкими, чтобы не мешать обслуживанию запросов от настоящих пользователей.

Обратите внимание на то, что при проверке готовности мы по-прежнему ставим вопрос ребром: да или нет. Просто наш ответ стал более информативным. Мы пытаемся как можно точнее определить, *довольны ли пользователи*.

Сервисы и предохранители

Как вы знаете, в случае провала проверки на *работоспособность* контейнер автоматически перезапускается с экспоненциальной выдержкой. Это мало чем поможет в ситуации, когда с самим контейнером все хорошо, а сбой произошел в одной из его зависимостей. При этом проверка *готовности* будет иметь такую семантику: «Со мной все в порядке, но в данный момент я не могу обслуживать пользовательские запросы».

В этой ситуации контейнер будет удален из любых сервисов, которые он обслуживает, а Kubernetes перестанет отправлять ему запросы до тех пор, пока тот снова не станет готовым к работе. Это более подходящая реакция на отказавшую зависимость.

Представьте, что у вас есть цепочка из десяти микросервисов, каждый из которых зависит от предыдущего (это значит, не может выполнять без него какую-то важную часть работы). И вот последний из них отказывает. Предпоследний сервис обнаруживает отказ и проваливает свою проверку готовности. Kubernetes его отключает, и об этом узнает следующий сервис, и так далее вверх по цепочке. В итоге отказывает клиентский сервис, и вы получаете оповещение от системы мониторинга методом черного ящика (в идеале).

Как только изначальная проблема будет улажена (или устранена за счет автоматической перезагрузки), все остальные сервисы в цепочке снова станут готовыми к работе, не требуя перезапуска и не теряя своего состояния. Это пример *шаблона «Предохранитель»* (martinfowler.com/bliki/CircuitBreaker.html). Когда приложение обнаруживает сбой в конце цепочки вызовов, оно выводит себя из эксплуатации (через проверку готовности), чтобы предотвратить поступление дальнейших запросов, и ждет, пока проблема не будет устранена.

Плавная деградация

Предохранитель помогает проблеме как можно быстрее себя проявить, однако вы должны проектировать свои сервисы так, чтобы отказ одного или нескольких компонентов не приводил к остановке всей системы. Пытайтесь привить своим

сервисам *плавную деградацию*: если они не могут справиться со всеми своими обязанностями, позвольте им делать хоть что-то.

В распределенных системах следует исходить из того, что сервисы, компоненты и соединения практически в любой момент могут отказать каким-то загадочным образом. Устойчивая система способна с этим справиться, не прекращая работу полностью.

Резюме

О мониторинге можно говорить очень долго. У нас нет возможности сказать все, что нам бы хотелось, но мы надеемся, что в этой главе вы нашли для себя полезную информацию о традиционных методиках мониторинга, их возможностях и ограничениях и о том, как их адаптировать для облачно-ориентированной среды.

Концепция *наблюдаемости* дает более широкую картину происходящего, чем традиционные журнальные файлы и проверки методом черного ящика. Важную часть этой картины составляют показатели, и в следующей — последней — главе мы с головой окунемся в эту тему в контексте Kubernetes.

Но прежде, чем листать дальше, давайте вспомним ключевые моменты.

- Мониторинг методом черного ящика позволяет обнаруживать предсказуемые сбои путем наблюдения за поведением системы извне.
- Распределенные системы раскрывают ограничения традиционного мониторинга, поскольку их состояние нельзя однозначно назвать *рабочим* или *нерабочим*: они постоянно находятся в режиме частичной деградации качества обслуживания. Иными словами, на борту корабля никогда ничего не идет в точности так, как нужно.
- Журнальные записи помогают в обнаружении проблемы, которая уже произошла, но их сложно масштабировать.
- Использование показателей открывает новое пространство возможностей, в котором все не сводится только к *«работает/не работает»*, и предоставляет вам поток числовых, хронологических данных о сотнях и тысячах характеристик вашей системы.
- Показатели помогают разбираться в *причинах* произошедшего и определять проблемные тенденции до того, как они приведут к перебоям в работе.
- Трассировка записывает события с точной хронологией на протяжении всего жизненного цикла отдельно взятого запроса, помогая вам диагностировать проблемы производительности.

- Наблюдаемость объединяет в себе традиционный мониторинг, ведение журнала, показатели, трассировку и любые другие способы получения информации о системе.
- Наблюдаемость также является шагом навстречу командной культуре в инженерии, основанной на фактах и обратной связи.
- Внешние проверки методом черного ящика по-прежнему играют важную роль, позволяя убедиться в доступности пользовательских сервисов. Но не пытайтесь создавать их с нуля — используйте сторонние системы мониторинга, такие как Uptime Robot.
- Девятки неважны, если пользователи недовольны.

16 Показатели в Kubernetes

> Можно так много узнать о предмете, что стать совершенно невежественным.
>
> *Фрэнк Герберт. Капитул Дюны*

В этой главе мы обратимся к концепции показателей, с которой познакомились в главе 15, и углубимся в детали: какие виды показателей существуют, какие из них важны в контексте облачно-ориентированных сервисов, как выбрать те показатели, на которые следует ориентироваться, как проанализировать собранные данные и на основании этого принять решение и как превратить необработанные показатели в полезные информационные панели и оповещения? В конце мы рассмотрим некоторые инструменты и платформы для работы с показателями.

Что на самом деле представляют собой показатели

Поскольку подход к наблюдаемости, ориентированный на показатели, является относительно новым в мире DevOps, стоит поговорить о том, что собой представляют показатели и как их лучше всего использовать.

Как мы уже видели в подразделе «Введение в показатели» на с. 334, показатели — это числовая мера определенных вещей. В качестве примера, знакомого из мира традиционных серверов, можно привести использование оперативной памяти на отдельном компьютере. Если процессам выделено всего 10 % физической памяти, у компьютера есть запасная емкость. Но если используется целых 90 %, компьютер, очевидно, очень загружен.

Таким образом, одним из видов ценной информации, которую нам могут дать показатели, является снимок того, что происходит в конкретный момент. Но мы можем пойти дальше. Потребление памяти постоянно колеблется в зависимости от запуска и остановки рабочих заданий, и нас может заинтересовать *изменение* этого значения во времени.

Хронологические данные

Если сведения о потреблении памяти собираются регулярно, вы можете построить из них *временной ряд*. На рис. 16.1 показан недельный график потребления памяти на узле Google Kubernetes Engine, и он дает куда более наглядную картину происходящего по сравнению с изолированными значениями.

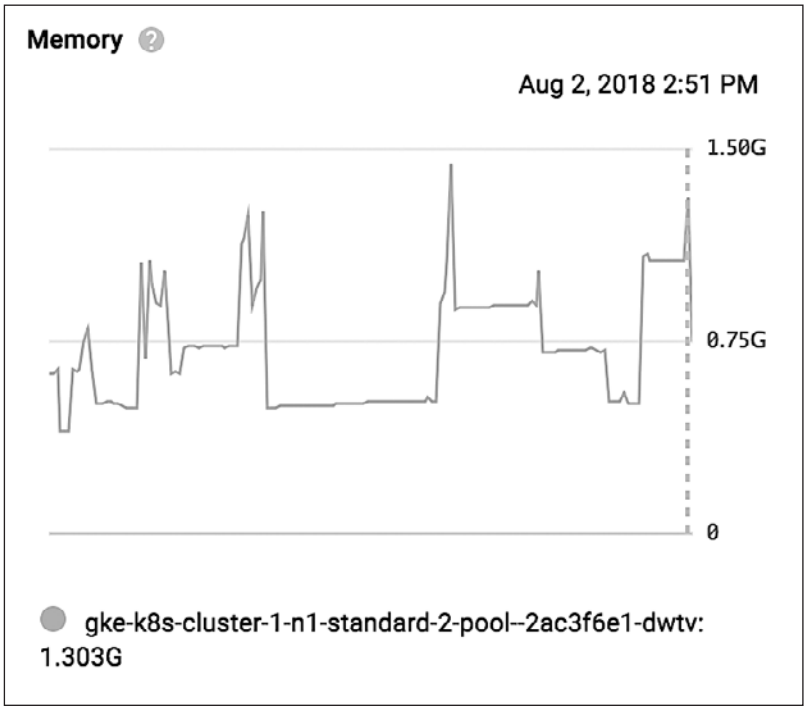

Рис. 16.1. Хронологический график использования памяти на узле GKE

Большинство показателей, которые интересуют нас в контексте облачно-ориентированной наблюдаемости, имеют хронологический характер. Кроме того, они являются числовыми. В отличие от, скажем, журнальных записей, с показателями можно производить математические и статистические операции.

Счетчики и измерители

Но что это за цифры? Некоторые величины можно выразить в виде целых чисел (например, количество физических процессоров на компьютере), но в большинстве случаев понадобится использовать десятичные значения. Поэтому, чтобы не приходилось иметь дело с несколькими типами значений, показатели почти всегда представлены в виде чисел с плавающей запятой.

С учетом этого показатели в основном бывают двух разновидностей: *счетчики* и *измерители*. Счетчики могут только увеличиваться (или сбрасываться в ноль) и подходят для измерения таких характеристик, как количество обработанных запросов или число полученных ошибок. Измерители же могут меняться в обе стороны и подходят для постоянно колеблющихся величин (например, для потребления памяти) или для выражения соотношений других значений.

На некоторые вопросы можно дать однозначный ответ — *да* или *нет*: например, реагирует ли конкретная конечная точка на HTTP-соединения. В этом случае больше подойдет измеритель с ограниченным диапазоном значений: скажем, 0 и 1.

HTTP-проверку конечной точки можно назвать `http.can_connect`: если точка отвечает, значение равно `1`, если нет — `0`.

О чем нам могут поведать показатели

Для чего нужны показатели? Как мы уже узнали ранее, по ним можно определить, что проблема есть. Например, если частота ошибок внезапно увеличивается (или наблюдается всплеск запросов к вашей странице поддержки), это указывает на некий сбой. Вы можете автоматически генерировать оповещения для определенных показателей на основе пороговых значений.

Но показатели также могут сказать вам о том, насколько хорошо все работает: например, сколько пользователей ваше приложение может выдержать одновременно. Долгосрочные тенденции в изменении этих значений могут пригодиться как для принятия решений в области системного администрирования, так и для бизнес-аналитики.

Выбор подходящих показателей

Сначала вы можете подумать: «Если показатели — это хорошо, то чем больше я их добавлю, тем лучше!» Но это так не работает. Вы не можете мониторить все подряд. Например, Google Stackdriver дает доступ буквально к сотням показателей облачных ресурсов, включая следующие:

- `instance/network/sent_packets_count` — количество сетевых пакетов, отправленных каждым вычислительным узлом;
- `storage/object_count` — общее количество объектов в каждом бакете хранилища;
- `container/cpu/utilization` — доля процессорных ресурсов, которую использует контейнер в текущий момент.

Это далеко (cloud.google.com/monitoring/api/metrics) не весь список. Если бы вы захотели вывести графики сразу для всех показателей, вам бы понадобился монитор размером с дом и вы ни за что не смогли бы извлечь из информации что-то полезное. Лучше *сфокусироваться* на особо интересующих показателях. На что же следует обращать внимание при наблюдении за собственным приложением? Только вы можете ответить на этот вопрос, однако у нас есть несколько советов, которые могут оказаться полезными. Далее в этом разделе мы рассмотрим некоторые распространенные шаблоны показателей для обеспечения наблюдаемости, нацеленные на разную аудиторию и созданные для удовлетворения разных требований.

Стоит отметить, что это идеальная возможность для совместной работы в стиле DevOps: размышления и разговоры о том, какие показатели вам нужны, следует начинать на старте разработки, а не в ее финале (см. подраздел «Учимся вместе» на с. 34).

Сервисы: шаблон RED

Большинство людей, использующих Kubernetes, запускают разного рода веб-сервисы: пользователи делают запросы, а приложение возвращает ответы. В роли *пользователей* могут выступать программы или другие сервисы. В распределенной системе, основанной на микросервисах, все сервисы отправляют запросы друг другу и используют результаты для выдачи информации другим сервисам. Как бы то ни было, это система, ориентированная на запросы.

Что следует знать о такой системе?

- Одним из очевидных ответов является количество *запросов*, которые вы получаете.
- Еще следует обращать внимание на количество запросов, которые по разным причинам оказались неудачными, — то есть на число *ошибок*.
- И третьим полезным показателем является *продолжительность* каждого запроса. Она дает представление о том, насколько хорошо работает ваш сервис и в какой степени пользователи могут быть недовольны.

Шаблон RED (Requests-Errors-Duration — «запросы — ошибки — продолжительность») — это классическая концепция наблюдаемости, которую начали применять еще на самых ранних этапах развития онлайн-сервисов. В книге *Site Reliability*

Engineering[1] от Google рассказывается о *четырех золотых сигналах*, которыми, в сущности, являются запросы, уровень ошибок, время отклика и степень загруженности (о ней чуть позже).

Инженер Том Уилки, придумавший акроним RED, дает обоснование этого шаблона в своей статье.

> Почему для каждого сервиса вы должны измерять одни и те же показатели? Ведь каждый сервис особенный, не так ли? Преимущества от одинакового обращения со всеми сервисами с точки зрения мониторинга состоят в масштабируемости команд системного администрирования. Если все сервисы на вид, на ощупь и вкус ничем не различаются, это снижает когнитивную нагрузку на тех, кто реагирует на происшествие. К тому же, если обращаться со всеми сервисами одним и тем же образом, многие повторяющиеся задачи можно будет автоматизировать.
>
> *Том Уилки*

Так как же измерять эти значения? Общее количество запросов может только увеличиваться, поэтому более правильным будет следить за их *частотой*: например, за количеством запросов в секунду. Это даст хорошее представление о том, сколько трафика система обрабатывает в заданный период времени.

Ошибки лучше считать в виде процента от количества запросов, так как эти две величины связаны между собой. Таким образом, информационная панель типичного сервиса может показывать:

- количество запросов, полученных в секунду;
- процент запросов, которые вернули ошибку;
- продолжительность запросов (известную также как *латентность*).

Ресурсы: шаблон USE

Как вы уже видели, шаблон RED предоставляет полезную информацию о качестве работы ваших сервисов и о том, как их воспринимают пользователи. Вы можете думать об этом как о нисходящем подходе к рассмотрению данных наблюдаемости.

Инженер Брендан Грегг, отвечающий за производительность в компании Netflix, разработал шаблон USE (www.brendangregg.com/usemethod.html), олицетворяющий восходящий подход. Такой подход призван помочь с анализом проблем, связанных

---

[1] Если вы не знакомы с SRE, почитайте замечательную книгу: *Бейер Б., Джоунс К., Петофф Д., Мёрфи Р.* Site Reliability Engineering. Надежность и безотказность как в Google. — СПб.: Питер, 2019.

с производительностью и поиском узких мест. USE означает Utilization, Saturation и Errors (использование, степень загруженности и ошибки).

Этот шаблон предназначен не для сервисов, а для *ресурсов* — компонентов физических серверов, таких как процессор и диски, или сетевых интерфейсов и каналов. Любой из них может оказаться узким местом в производительности системы, и выявить их помогут показатели USE.

- *Эффективность использования*. Среднее время использования ресурса для обслуживания запросов или доля емкости ресурса, которая сейчас занята. Например, потребление диска, у которого занято 90 % места, будет равно 90 %.
- *Насыщенность*. Степень перегруженности ресурса или длина очереди запросов, которые ждут, когда ресурс станет доступным. Например, если выполнения на процессоре ждут десять процессов, степень загруженности процессора будет равна десяти.
- *Ошибки*. Сколько раз завершилась неудачей операция с этим ресурсом. Например, у диска с неисправными секторами количество ошибок может равняться 25 неудачным операциям чтения.

Измерение этих показателей для ключевых ресурсов в вашей системе является хорошим способом поиска узких мест. Ресурсы с низкой степенью потребления, загруженности и ошибок, скорее всего, в порядке. Но все, что отклоняется от нормы, заслуживает вашего внимания. Например, если какой-то сетевой канал является загруженным или имеет большое количество ошибок, он может быть одной из причин проблем с общей производительностью.

> Метод USE — это простая стратегия, с помощью которой вы можете произвести полную проверку работоспособности системы и найти типичные узкие места и ошибки. Ее можно применять на ранних этапах проверки, чтобы быстро выделить проблемные области и при необходимости исследовать их более детально с использованием других методик.
>
> Сильной стороной шаблона USE являются его скорость и наглядность: вы вряд ли пропустите какие-то проблемы, просматривая все ресурсы. Однако он рассчитан на поиск только определенного рода проблем, узких мест и ошибок, поэтому должен быть лишь одним из многих инструментов в вашем арсенале.
>
> *Брендан Грегг*

Бизнес-показатели

Итак, мы рассмотрели показатели уровня приложений и сервисов (см. подраздел «Сервисы: шаблон RED» на с. 349), которые в первую очередь интересны разра-

ботчикам. Мы также уделили внимание аппаратным показателям (см. подраздел «Ресурсы: шаблон USE» на с. 350), полезным для системных администраторов и инженеров инфраструктуры. Но что насчет бизнеса? Может ли наблюдаемость помочь руководителям разных уровней понять, насколько хорошо идут дела, и повлиять на принятие бизнес-решений? И какие показатели в этом могут участвовать?

Большинство компаний уже отслеживают интересующие их ключевые показатели эффективности (key performance indicators, KPI), такие как доход от продаж, рентабельность и затраты на привлечение клиентов. Эти сведения обычно предоставляются финансовым отделом и не требуют поддержки от разработчиков и тех, кто занимается инфраструктурой.

Однако приложения и сервисы тоже могут сгенерировать полезные бизнес-показатели. Например, если у компании есть продукт, доступный в виде подписки (такой как *программное обеспечение как услуга* (software-as-a-service, SaaS)), полезной будет следующая информация о его подписчиках.

- Анализ воронки продаж (сколько людей заходит на целевую страницу, сколько щелчков кнопкой мыши нужно, чтобы дойти до страницы регистрации, сколько посетителей завершили транзакцию и т. д.).
- Соотношение регистраций и отмен (коэффициент оттока).
- Доход от каждого клиента (полезно для вычисления ежемесячной выручки, среднего дохода с одного клиента и пожизненной ценности клиента).
- Эффективность страниц помощи и поддержки (например, процент людей, которые утвердительно ответили на вопрос: «Решила ли эта страница вашу проблему?»).
- Трафик к странице состояния системы (всплеск которого часто наблюдается во время перебоев в работе или ухудшения качества обслуживания).

Большую часть этой информации будет легче визуализировать, если приложение само сгенерирует показатели в режиме реального времени, чтобы вам не нужно было анализировать их по факту, обрабатывая журнальные записи и обращаясь к базам данных. Когда вы подготавливаете свои приложения для генерации показателей, не пренебрегайте информацией, которая может быть важна для бизнеса.

Сложно провести четкую границу между данными наблюдаемости, нужными бизнесу и специалистам по привлечению клиентов, и теми, которые могут понадобиться техническим специалистам. На самом деле эти данные во многом пересекаются. Обсуждение показателей разумно проводить на ранних стадиях с участием

всех заинтересованных сторон: вы должны согласовать, какую информацию нужно собирать, как часто это следует делать, как ее агрегировать и т. д.

Тем не менее эти две (как минимум) группы людей имеют разные вопросы к данным наблюдаемости, которые вы собираете, поэтому каждой из них потребуется отдельное их представление. Вы можете использовать общее *озеро данных* для создания информационных панелей (см. раздел «Визуализация показателей с помощью информационных панелей» на с. 362) и отчетов для всех вовлеченных сторон.

Показатели Kubernetes

В общих чертах мы обсудили наблюдаемость и показатели, а также рассмотрели разные типы данных и способы их анализа. Но как это все применить к Kubernetes? Какие показатели стоит отслеживать в ваших кластерах и какого рода решения они могут помочь вам принять?

На самом низком уровне находится инструмент под названием cAdvisor, который наблюдает за использованием ресурсов и собирает статистику производительности для контейнеров, запущенных на каждом узле кластера: например, сколько ресурсов процессора, памяти и дискового пространства использует контейнер. cAdvisor является частью Kubelet.

Данные cAdvisor потребляет платформа Kubernetes и использует для этого kubelet. Исходя из полученной информации, она принимает решения о планировании, автомасштабировании и т. д. Но данные также можно экспортировать в сторонний сервис, который будет их визуализировать и отправлять вам уведомления. Например, было бы полезно следить за тем, сколько ресурсов процессора и памяти расходует каждый контейнер.

Вы также можете мониторить и саму платформу Kubernetes, используя инструмент под названием `kube-state-metrics`. Он следит за Kubernetes API и генерирует информацию о таких логических объектах, как узлы, pod-оболочки и развертывания. Эти данные могут пригодиться и для обеспечения наблюдаемости кластера. Например, если для развертывания, которое временно не может быть запланировано по той или иной причине (возможно, кластеру не хватает емкости), выделены какие-то реплики, вам, наверное, следует об этом знать.

Как обычно, проблема не в нехватке показателей, а в выборе тех из них, которые стоит отслеживать и визуализировать. Далее дадим несколько советов по этому поводу.

Показатели работоспособности кластера

Чтобы следить за работоспособностью и производительностью своего кластера на самом высоком уровне, вы должны обращать внимание как минимум на следующее:

- количество узлов;
- состояние работоспособности узла;
- среднее количество pod-оболочек на одном узле и в целом;
- среднее выделение/использование ресурсов на одном узле и в целом.

Эти общие показатели помогут вам понять, насколько хорошо работает кластер, достаточно ли у него емкости, как меняется со временем его загруженность и следует ли его расширить/сократить.

Если вы используете управляемый сервис Kubernetes, такой как GKE, неработоспособные узлы будут обнаруживаться и восстанавливаться автоматически (при условии, что вы включили автовосстановление для своего кластера и пула узлов). Но вам все равно не помешает отслеживать отклонения в частоте отказов, что может свидетельствовать о внутренних проблемах.

Показатели развертывания

В отношении развертываний стоит следить за такими данными, как:

- количество развертываний;
- количество сконфигурированных реплик на каждом развертывании;
- количество недоступных реплик на каждом развертывании.

Особенно полезной будет возможность отслеживать эту информацию во времени при включении разных параметров автомасштабирования Kubernetes (см. пункт «Автомасштабирование» на с. 145). В частности, данные о недоступных репликах могут предупредить вас о проблемах с емкостью.

Показатели контейнера

Самая полезная информация на уровне контейнера, которую нужно знать:

- среднее количество контейнеров/pod-оболочек на одном узле и в целом;
- соотношение потребленных ресурсов и запросов/лимитов контейнера (см. подраздел «Запросы ресурсов» на с. 108);

- работоспособность/готовность контейнеров;
- количество перезапусков контейнеров/pod-оболочек;
- входящий/исходящий трафик и ошибки для каждого контейнера.

Поскольку Kubernetes автоматически перезапускает контейнеры, которые столкнулись с проблемами или исчерпали свои лимиты на ресурсы, вам необходимо знать, как часто это происходит. Чрезмерное количество перезапусков может свидетельствовать о проблеме с конкретным контейнером. Если контейнер регулярно исчерпывает свои лимиты на ресурсы, это может быть признаком программной ошибки — или же просто следует немного поднять эти лимиты.

Показатели приложения

Какие бы языки или программные платформы ни использовало приложение, вероятно, имеется и библиотека или инструмент для экспорта из него пользовательских показателей. Эта информация особенно полезна разработчикам и системным администраторам, которые с ее помощью могут увидеть, какие действия выполняет приложение, как часто оно это делает и сколько на это уходит времени. Таковы ключевые индикаторы проблем с производительностью или доступностью.

Какие показатели приложения следует собирать и экспортировать, зависит от того, чем именно оно занимается. Но существуют общепринятые методы. Например, если ваш сервис потребляет сообщения из очереди, обрабатывает и предпринимает на их основе какие-то действия, вам, вероятно, стоит знать следующее:

- количество полученных сообщений;
- количество успешно обработанных сообщений;
- количество некорректных или ошибочных сообщений;
- время, необходимое для обработки каждого сообщения и реакции на него;
- количество успешных действий, которые были сгенерированы;
- количество неудачных действий.

Аналогично, если ваше приложение в основном ориентировано на запросы, вы можете использовать шаблон RED (см. подраздел «Сервисы: шаблон RED» на с. 349):

- сколько запросов получено;
- сколько ошибок возвращено;
- продолжительность (время обработки каждого запроса).

На ранних этапах разработки бывает непросто определить, какие показатели окажутся полезными. Если сомневаетесь, записывайте все — это не так уж и сложно. Однажды вы можете обнаружить непредвиденную проблему в промышленной среде благодаря данным, которые сейчас кажутся незначительными.

> Если что-то двигается, отобразите это в виде графика. Но даже если нет, все равно пусть график будет на случай, если однажды оно начнет двигаться.
>
> *Лори Диннесс (Bloomberg)*[1]

Если вы собираетесь генерировать в своем приложении бизнес-показатели (см. подраздел «Бизнес-показатели» на с. 351), их можно вычислять и экспортировать так же, как и пользовательские.

Еще одна полезная с точки зрения бизнеса вещь — возможность оценивать работу вашего приложения в соответствии с целевым уровнем качества обслуживания (Service Level Objectives, SLO) или соглашениями об уровне качествауслуг (Service Level Agreements, SLA), которые вы могли заключить с клиентами. Вам также пригодится сравнение SLO с эффективностью предоставления услуг поставщиком. Вы можете создать пользовательский показатель, чтобы изобразить желаемое время обработки запроса (например, 200 миллисекунд) и вывести поверх него ту производительность, которая имеется на текущий момент.

Показатели среды выполнения

На уровне среды выполнения большинство библиотек для работы с показателями также предоставляют полезные данные о том, чем занимается программа. Например, такие:

- количество процессов/потоков/горутин;
- использование кучи и стека;
- использование памяти вне кучи;
- пулы буферов сетевого ввода/вывода;
- периоды работы и бездействия сборщика мусора (для языков, где таковой имеется);
- количество используемых файловых дескрипторов или сетевых сокетов.

Такого рода информация может быть крайне полезной при диагностике низкой производительности или даже сбоев в работе. Например, долгоработающие при-

[1] twitter.com/lozzd/status/604064191603834880.

ложения со временем часто увеличивают потребление памяти до тех пор, пока не исчерпают лимиты на ресурсы Kubernetes (в результате чего удаляются и запускаются заново). Показатели среды выполнения, особенно в сочетании с данными о работе приложения, укажут, где именно происходит эта утечка памяти.

Итак, вы получили представление о том, какие показатели стоит собирать. В следующем разделе вы увидите, что *делают* с этими данными, — иными словами, как их анализируют.

Анализ показателей

Данные — это не то же самое, что информация. Чтобы получить полезную информацию из собранных необработанных данных, их нужно агрегировать, обработать и проанализировать — это значит собрать *статистику*. Статистика полна нюансов, особенно если говорить о ней абстрактно, поэтому обратимся к реальному примеру измерения продолжительности запроса.

В подразделе «Сервисы: шаблон RED» на с. 349 мы упоминали о необходимости отслеживания показателя продолжительности запросов к сервису, но не сказали о том, как именно это сделать. Что конкретно мы понимаем под продолжительностью? Обычно нам интересно то, сколько времени пользователю приходится ждать ответа на свой запрос.

Например, для сайта мы можем определить продолжительность как время между тем, когда пользователь подключается к серверу, и тем, когда сервер возвращает ему ответ (на самом деле общее время ожидания пользователя длиннее, поскольку в него входит и время на создание соединения, чтение ответа и вывод его в браузере. Но обычно у нас нет доступа к этим данным, поэтому мы собираем то, что можем).

Но у каждого запроса своя продолжительность — как же мы агрегируем данные для сотен или даже тысяч запросов в единое число?

Что не так с простым средним значением

Очевидным решением является вычисление среднего значения. Но если немного подумать, понятие *средней величины* не такое уж и простое. У статистиков в ходу старая шутка о том, что у среднего человека чуть меньше двух ног. Иными словами, у большинства людей количество ног выше среднего. Как такое может быть?

Почти у всех у нас есть две ноги, но вместе с тем встречаются одноногие и безногие люди, что понижает общее среднее (возможно, кто-то имеет больше двух ног, но

их количество несравнимо ниже). Простое среднее значение не очень помогает разобраться в том, как распределены ноги среди населения или сколько ног у большинства людей.

Кроме того, бывает несколько видов средних значений. Вы, наверное, знаете, что в общеупотребительном смысле имеется в виду *математическое среднее*: сумма всех чисел во множестве, разделенная на их количество. Например, средний возраст группы из трех человек равен сумме их возрастов, разделенной на 3.

С другой стороны, существует *медиана*. Это значение, которое делит множество на две равные части: в одной значения больше медианы, а в другой — меньше. Например, если взять любую группу людей, половина из них по определению будет выше медианного роста, а другая половина — ниже.

Средние значения, медианы и выбросы

Что не так с вычислением обычной (математической) средней продолжительности запроса? Одной из важных проблем является то, что такая величина легко искажается из-за *выбросов*: одного или нескольких крайних значений, которые сильно влияют на средний показатель.

Таким образом, для усреднения показателей лучше подходит медиана, которая не настолько подвержена влиянию выбросов. Если усредненное (медианное) время отклика сервиса составляет одну секунду, у половины ваших пользователей оно будет меньше одной секунды, а у другой половины — больше.

На рис. 16.2 показано, насколько обманчивыми могут быть средние значения. Все четыре набора данных имеют одно и то же среднее значение, но в графическом представлении выглядят совсем по-разному (тем, кто занимается статистикой, этот пример знаком под названием *«Квартет Энскомба»*). Это еще и хорошая демонстрация того, насколько важной бывает визуализация данных по сравнению с просмотром необработанных чисел.

Вычисление перцентилей

Обсуждая показатели для наблюдения за системами, ориентированными на запросы, мы обычно заинтересованы в том, чтобы узнать, каков наихудший показатель времени отклика для пользователей, а не ее среднее значение. В конце концов, тот факт, что общее усредненное время отклика составляет одну секунду, не является утешением для тех немногих пользователей, которым приходится ждать ответа по десять и более секунд.

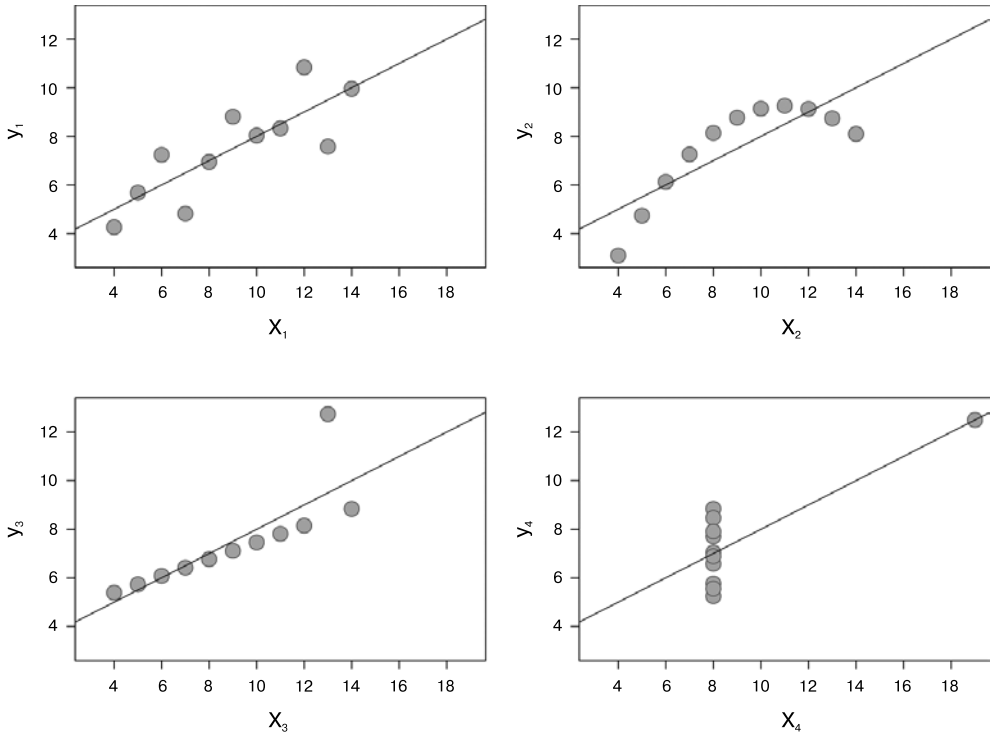

Рис. 16.2. Все четыре набора данных имеют одно и то же (математическое) среднее значение (рис. Schutz, CC BY-SA 3.0)

Чтобы получить подобную информацию, можно разбить данные на *перцентили*. Время отклика с 90-м перцентилем (часто обозначают как *P90*) — значение, которое выше, чем у 90 % ваших пользователей. Иными словами, 10 % пользователей будут иметь время отклика, которое выше, чем значение P90.

Используя эту терминологию, медиана — это 50-й перцентиль, или P50. Также в области наблюдаемости часто измеряют перцентили P95 и P99.

Применение перцентилей к показателям

Игорь Видлер из Travis CI замечательно продемонстрировал (igor.io/latency), что это означает на практике. Для начала был взят набор из 135 000 запросов к промышленному сервису, выполненных за десять минут (рис. 16.3). Как можете убедиться сами, данные содержат много шума и скачков, и в таком виде из них сложно сделать какие-то полезные выводы.

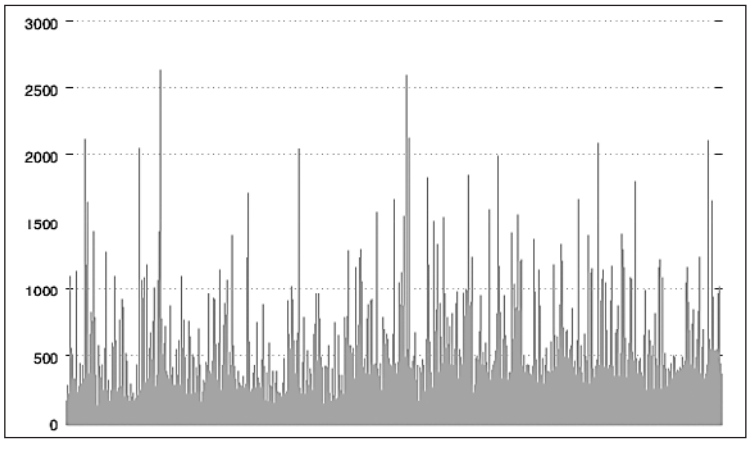

Рис. 16.3. Необработанные данные о времени отклика для 135 000 запросов, в миллисекундах

Теперь посмотрим, что произойдет, если мы усредним эти данные на десятисекундных интервалах (рис. 16.4). Результат выглядит прекрасно: все значения находятся ниже 50 миллисекунд. Получается, что у большинства наших пользователей время отклика не превышает 50 миллисекунд. Но так ли это?

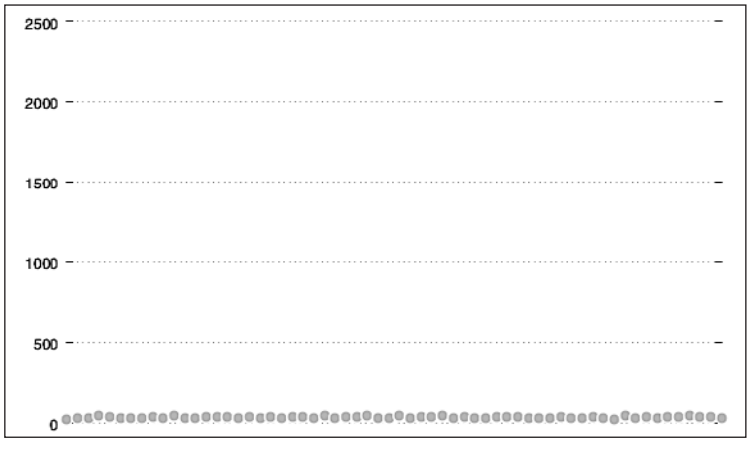

Рис. 16.4. Среднее время отклика для тех же данных на десятисекундных интервалах

Изобразим перцентиль P99. Это максимальное наблюдаемое время отклика с отсечением 1 % самых больших значений. Результат выглядит совсем иначе (рис. 16.5). Мы видим неровный график, где большинство значений концентрируются между 0 и 500 миллисекундами, и только несколько из них находятся в районе 1000 миллисекунд.

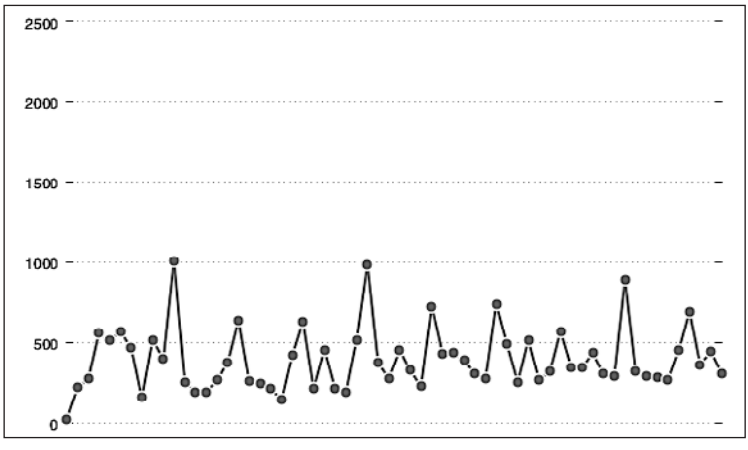

Рис. 16.5. Время отклика с 99-м перцентилем для тех же данных

Обычно нас интересуют наихудшие показатели

Поскольку люди чаще всего заостряют свое внимание на медленных веб-запросах, данные P99, скорее всего, дадут более реалистичную картину латентности, с которой пользователи имеют дело. Представьте, к примеру, высоконагруженный сайт с миллионом просмотров страниц в день. Если время отклика P99 составляет десять секунд, следовательно, 10 000 запросов к страницам обрабатывались дольше десять секунд. А это означает много недовольных пользователей.

Но все еще хуже: в распределенных системах каждый просмотр страницы может потребовать десятка или даже сотни внутренних запросов. Если время отклика P99 для каждого внутреннего сервиса равно десяти секундам, а на просмотр одной страницы уходит десять внутренних запросов, количество медленных просмотров увеличивается до 100 000 в день. То есть 10 % пользователей недовольны, а это уже серьезная проблема (engineering.linkedin.com/performance/who-moved-my-99th-percentile-latency).

Не перцентилями едиными

У реализации перцентильной латентности во многих сервисах измерения показателей есть одна проблема: данные о запросах собираются локально, а статистика агрегируется централизованно. Следовательно, вы получаете среднюю латентность P99, собранную каждым агентом, количество которых может измеряться сотнями.

Перцентиль и так является усреднением, и, усредняя средние значения, мы попадаем в известную статистическую ловушку (ru.wikipedia.org/wiki/Парадокс_Симпсона). Результат может отличаться от реального среднего.

В зависимости от того, как мы решим агрегировать данные, итоговая величина латентности P99 может отклоняться на порядок, что не очень способствует вычислению полезного результата. Если ваш сервис сбора показателей не учитывает каждое сырое событие и не выдает настоящее среднее значение, эта величина будет ненадежной.

Инженер Ян Куй (medium.com/theburningmonk-com/we-can-do-better-than-percentile-latencies-2257d20c3b39) считает, что лучше следить за *отклонениями* от нормы, чем за самой *нормой*.

> Что можно использовать вместо перцентилей в качестве основного показателя при мониторинге производительности приложения с возможностью отправлять уведомления, когда ситуация начнет ухудшаться?
>
> Посмотрев на свои SLO и SLA, вы наверняка заметите, что они будут указывать примерно на следующее: «99 % запросов должны завершаться за секунду или быстрее». Иными словами, менее чем 1 % запросов позволено завершиться более чем за секунду.
>
> А что, если вместо этого мы будем отслеживать процент запросов, превышающих пороговое значение? Чтобы узнавать о нарушениях SLA, мы можем генерировать уведомление, когда этот показатель превышает 1 % в какой-то заранее определенный отрезок времени.
>
> *Ян Куй*

Если каждый агент предоставляет показатель общего числа запросов и количества запросов, превысивших лимит, мы *можем* корректно усреднить эти данные, чтобы получить перцентиль запросов, нарушающих SLO (и сгенерировать уведомление).

Визуализация показателей с помощью информационных панелей

Из этой главы вы узнали, в чем польза показателей и какие из них следует записывать. Вы также познакомились с некоторыми практичными статистическими методиками группового анализа данных. Все замечательно, но что мы собираемся *делать* с полученными показателями?

Ответ прост: мы будем их визуализировать, объединять в информационные панели и, возможно, генерировать на их основе уведомления. Об уведомлениях поговорим в следующем разделе, а пока что рассмотрим некоторые инструменты и методики для визуализации и создания информационных панелей.

Использование стандартной компоновки для всех сервисов

Если у вас не несколько сервисов, а гораздо больше, для каждого из них имеет смысл использовать одну и ту же компоновку информационных панелей. Тогда тот, кто отвечает на вызов, сможет открыть веб-консоль проблемного сервиса и сразу же понять, как ее интерпретировать, даже ничего не зная о самом сервисе.

Том Уилки в блоге Weaveworks (www.weave.works/blog/the-red-method-key-metrics-for-microservices-architecture) предлагает следующий стандартный формат (рис. 16.6):

- в каждой строке указывается один сервис;
- данные о частоте запросов и ошибок располагаются слева, частота ошибок представлена в виде процента от числа запросов;
- показатель времени отклика находится справа.

Рис. 16.6. Компоновка информационной панели сервисов, предлагаемая компанией Weaveworks

Не обязательно использовать именно эту компоновку — главное, чтобы информационные панели были выполнены в едином стиле, с которым знакомы все сотрудники. Вы должны регулярно просматривать свои ключевые панели (как минимум еженедельно), анализируя данные за прошедшую неделю, чтобы все знали, как выглядят *нормальные* показатели.

Информационная панель, отображающая *количество запросов*, *уровень ошибок* и *продолжительность*, хорошо подходит для сервисов (см. подраздел «Сервисы: шаблон RED» на с. 349). Для ресурсов, таких как узлы кластера, диски и сеть, самыми полезными показателями обычно являются *потребление*, *степень загруженности* и *уровень ошибок* (см. подраздел «Ресурсы: шаблон USE» на с. 350).

Информационный излучатель на основе обобщенных панелей данных

Наличие сотни сервисов подразумевает и столько же информационных панелей. Если это ваш случай, вы вряд ли часто будете в них заглядывать. Конечно, доступ к этим данным очень важен (например, они помогают понять, какой из сервисов отказывает), но в таких масштабах требуется более общее представление.

Поэтому было бы неплохо создать обобщенную информационную панель, которая отобразит данные о запросах, ошибках и продолжительности для *всех* ваших сервисов в совокупности. Не делайте ничего необычного вроде диаграмм с областями и накоплением — ограничьтесь простыми графиками общего количества запросов, общей доли ошибок и общего времени отклика. Это более точные представления по сравнению со сложными диаграммами. К тому же их легче интерпретировать.

В идеале следует использовать *информационный излучатель* (известный также как информационная стена или «большой наглядный график»). Это большой монитор, отображающий ключевые данные наблюдаемости так, чтобы они были видны всем заинтересованным группам сотрудников. Цель информационного излучателя состоит в том, чтобы:

- показать текущее состояние системы так, чтобы было понятно с одного взгляда;
- дать четкое представление о том, какие показатели важны;
- демонстрировать, какова *норма*.

Что следует выводить на экран? Только жизненно важную информацию, имеющую *большое значение* и отражающую *самочувствие* системы.

Хорошим примером являются мониторы, которые можно увидеть рядом с больничной койкой. Они выводят ключевые для человека показатели: сердцебиение, артериальное давление, насыщение крови кислородом, температуру и частоту дыхания. Существует множество других показателей, которые можно отслеживать у пациента, но на мониторы выводятся только те, что являются ключевыми на уровне обобщенной информационной панели. Любая серьезная проблема со здоровьем отразится на одном или нескольких этих показателях, а все остальное — вопрос диагностики.

Точно так же информационный излучатель должен показывать жизненно важную информацию о вашем бизнесе или сервисе. Если на нем отображаются показатели в виде чисел, их количество, вероятно, следует ограничить четырьмя или пятью. То же касается и графиков: их должно быть не больше четырех или пяти.

Может возникнуть соблазн втиснуть в информационную панель слишком много данных, в результате чего она будет выглядеть сложной и перегруженной. Это не то, к чему следует стремиться. Вы должны сосредоточиться на нескольких ключевых элементах и сделать так, чтобы их отовсюду было хорошо видно (рис. 16.7).

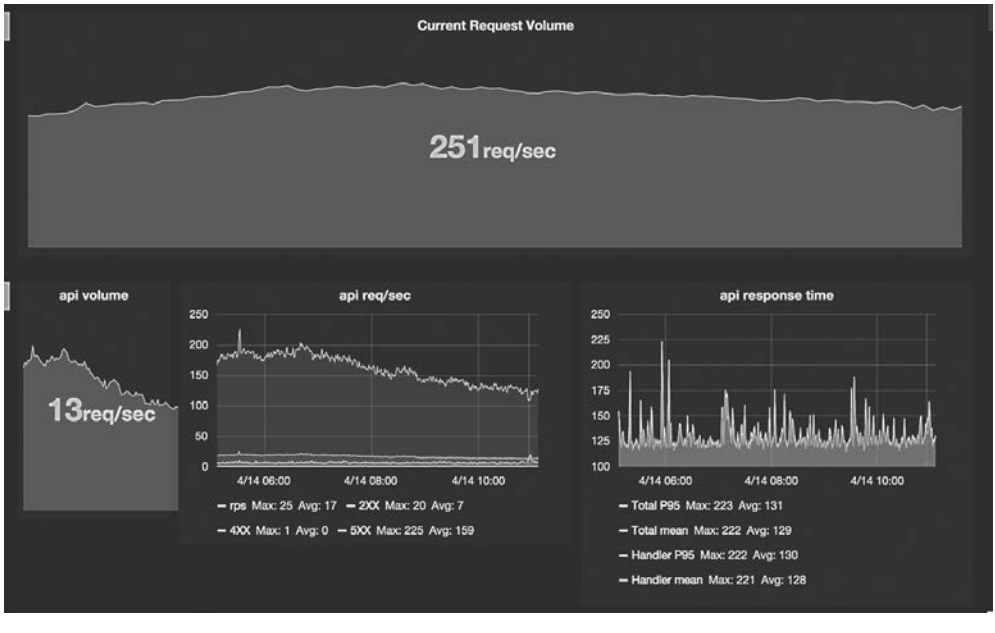

Рис. 16.7. Пример информационного излучателя, сгенерированного в Grafana Dash Gen (github.com/uber/grafana-dash-gen)

Отслеживайте то, что может сломаться

Помимо главного информационного излучателя и панелей для каждого отдельного сервиса и ресурса, стоит, наверное, создать и панели с определенными показателями, описывающими важные характеристики системы. Возможно, вам удастся выбрать некоторые из них, исходя из своей архитектуры. Но еще очень полезно иметь информацию о том, *что может сломаться*.

Каждый раз, когда происходит какое-то происшествие или перебой в работе, обращайте внимание на показатели или их комбинации, которые могли бы заранее предупредить об этой проблеме. Например, если перебой в работе промышленной среды вызван тем, что у сервера закончилось свободное место на диске, диаграмма дискового пространства могла бы заблаговременно предупредить вас о том, что доступное пространство на диске приближается к опасному уровню.

Мы не имеем в виду проблемы, которые возникают в течение минут или часов, — они обычно обнаруживаются системой автоматических уведомлений (см. раздел «Уведомления о показателях» ниже). Речь, скорее, о медленно надвигающихся «айсбергах», которые становятся заметными за несколько дней или недель до столкновения, и если их не увидеть вовремя и не предпринять маневр уклонения, ваша система «утонет» в самый неподходящий момент.

После инцидента всегда задавайтесь вопросом: «Что могло бы нас заранее о нем предупредить, знай мы, куда смотреть?. Если ответом является фрагмент данных, который у вас уже был, но вы не обращали на него внимания, примите меры, чтобы его как-то выделить. Информационная панель — один из способов это сделать.

Хотя уведомления могут сообщить о том, что некоторые значения достигли заданного порога, но в каких-то случаях об уровне опасности нельзя судить заранее. Диаграмма позволяет визуализировать колебания этого значения на протяжении длительного периода времени и помогает обнаружить проблемные тенденции до того, как они начнут сказываться на работе системы.

Уведомления о показателях

Вы, наверное, не ожидали, что мы потратим большую часть этой главы на обсуждение наблюдаемости и мониторинга, не упоминая уведомления. Для некоторых людей уведомления — это то, к чему сводится весь мониторинг. Мы считаем, что подобную философию пора менять, и на то есть целый ряд причин.

Что не так с уведомлениями

Уведомления указывают на какое-то неожиданное отклонение от стабильного рабочего состояния. Но ведь у распределенных систем таких состояний не бывает!

Как уже упоминалось, крупномасштабные распределенные системы никогда не являются полностью *рабочими*, они почти непрерывно находятся в состоянии частичной деградации качества обслуживания (см. пункт «Облачно-ориентированные приложения никогда не являются рабочими» на с. 331). У них есть огромное количество показателей, и если генерировать уведомления всякий раз, когда один из них выходит за пределы нормы, вы будете получать сотни бесполезных сообщений каждый день.

> Люди получают слишком много уведомлений из-за некачественной наблюдаемости и не доверяют своим инструментам настолько, чтобы позволить им отлаживать и диагностировать проблемы надежным образом. Поэтому они получают десятки и сотни сообщений, пытаясь найти в них зацепки о том, какова же причина проблемы. Таким образом, они просто гадают на кофейной гуще. В хаотичном будущем, навстречу которому мы все вместе несемся, необходимо стремиться к тому, чтобы число уведомлений было как можно меньшим, а не большим. Частота запросов, время отклика, доля ошибок, степень загруженности.
>
> *Чарити Мэйджорс*

Для некоторых несчастных людей получение множества дежурных уведомлений является привычным делом и даже образом жизни. Но в этом нет ничего хорошего, и не только по очевидным физиологическим причинам. Усталость, вызываемая большим количеством уведомлений, — хорошо известный феномен в медицине: медики констатируют понижение чувствительности к постоянным сигналам тревоги, вследствие чего повышается вероятность того, что при возникновении серьезной проблемы люди ее не заметят.

Чтобы система мониторинга была полезной, доля информационного шума в ней должна быть очень небольшой. Ложные срабатывания являются не просто досадными, но и опасными: они понижают доверие к системе и настраивают людей на то, что уведомления можно спокойно игнорировать.

Чрезмерные, непрекращающиеся и малозначительные уведомления были основным фактором аварии на АЭС «Три-Майл-Айленд» (humanisticsystems.com/2015/10/16/fit-for-purpose-questions-about-alarm-system-design-from-theory-and-practice). Даже если отдельные сообщения составлены как следует, их одновременная отправка в больших количествах может ошеломить системных администраторов.

Уведомление должно означать одну очень простую вещь: *прямо сейчас требуются действия со стороны человека* (www.infoworld.com/article/3268126/beware-the-danger-of-alarm-fatigue-in-it-monitoring.html).

Если действовать не нужно, не нужны и уведомления. Если что-то следует предпринять, но не *сейчас*, уведомление можно послать по электронной почте или в чате. А если с проблемой может справиться автоматизированная система, доверьте это ей: не будите ценного человека.

Дежурство не должно быть пыткой

Быть на постоянной связи со своими сервисами — это ключевой аспект философии DevOps, но не менее важно сделать это взаимодействие как можно менее болезненным.

Уведомления должны приходить редко, в исключительных ситуациях. На этот случай вам следует иметь устоявшуюся и эффективную процедуру, которая делает ситуацию как можно менее стрессовой.

Никто не должен находиться на дежурстве постоянно. При необходимости привлеките больше людей и разделите их на смены. Дежурному не обязательно быть специалистом в соответствующей области: его основная задача — классифицировать проблему, решить, требует ли она принятия каких-то мер, и передать информацию дальше.

Нагрузка на дежурных должна быть распределена равномерно, однако каждый имеет свои личные обстоятельства. Если у человека есть семья, если он занят чем-то еще вне работы, ему может быть не очень просто дежурить посменно. Организация данного аспекта работы справедливым по отношению ко всем образом требует осторожного и чуткого подхода.

Если дежурство является частью работы, это должно быть четко оговорено при найме. То, насколько часто и в каких обстоятельствах будут назначаться смены, следует прописать в контракте. Было бы нечестно предложить кому-то обычный восьмичасовой рабочий день, а затем потребовать, чтобы работник был доступен для вызова по ночам и в выходные дни.

Дежурства должны как следует компенсироваться надбавками в зарплате, отгулами или другими разумными вознаграждениями. И неважно, получали вы уведомления или нет: когда вы на дежурстве, вы находитесь на работе.

Кроме того, время нахождения на дежурстве должно быть строго ограничено. Менее занятые и более энергичные сотрудники могут вызваться добровольно,

чтобы снизить нагрузку на своих коллег, и это прекрасно, но следите за тем, чтобы никто не брал на себя слишком много.

Вы должны понимать, что, назначая людей дежурными, вы тратите человеческие ресурсы. Делайте это с умом.

Уведомления неотложные, важные и требующие принятия мер

Если уведомления — это так плохо, зачем мы вообще их обсуждаем? Дело в том, что без них не обойтись. Что-то может пойти не так — взорваться, упасть, остановиться, и, как правило, в самый неподходящий момент.

Наблюдаемость — прекрасная концепция, но проблему нельзя найти, если не искать. Использование информационных панелей является отличной идеей, но вы не платите людям за то, чтобы они круглые сутки следили за показателями. Если речь идет об обнаружении сбоев в работе или других проблем, происходящих прямо сейчас, и при этом требуется привлечь внимание сотрудников, автоматические уведомления, основанные на пороговых значениях, вне конкуренции.

Например, вы можете сделать так, чтобы система посылала уведомление, когда частота ошибок для заданного сервиса превысит 10 % в какой-то отрезок времени (скажем, пять минут). Или можете генерировать уведомление, когда время отклика P99 превысит определенное значение, такое как 1000 миллисекунд.

В целом, если проблема имеет реальное или потенциальное воздействие на рабочий процесс и меры должны быть предприняты незамедлительно живым человеком, уведомления желательно получать.

Не шлите уведомления о каждом показателе. Из сотен или даже тысяч выберите только небольшую их группу, которая может генерировать уведомления. Но, даже если уведомление появилось, не во всех случаях его стоит кому-то отсылать: только в ситуациях, когда проблемы являются *неотложными*, *важными* или *требующими принятия мер*.

- Важными, но несрочными уведомлениями можно заниматься в обычное рабочее время. Только то, что нельзя отложить до утра, должно доходить до дежурного.
- Неотложные, но неважные уведомления не стоят того, чтобы кого-то будить. Например, отказ малоиспользуемого внутреннего сервиса не затрагивает клиентов.
- Если для решения проблемы нельзя предпринять никаких незамедлительных мер, людей об этом уведомлять бессмысленно.

Во всех остальных случаях лучше отправлять асинхронные сообщения — по электронной почте, через Slack, обращаясь в службу поддержки, создавая заявку в системе управления проектом и т. д. Если в вашей организации рабочий процесс налажен правильно, их увидят и своевременно отреагируют. Не нужно вызывать у человека всплеск уровня кортизола (гормона стресса), поднимая посреди ночи громким уведомлением.

Отслеживайте свои уведомления и их последствия

Работники для вашей инфраструктуры важны не меньше, а может, и больше, чем облачные серверы и кластеры Kubernetes. Они наверняка и дороже обходятся, и уж точно их сложнее заменить. Таким образом, для них мониторинг имеет такой же смысл, как и для ваших сервисов.

Число уведомлений, сгенерированных за неделю, является хорошим показателем работоспособности и стабильности вашей системы. А по количеству вызовов, особенно после работы, на выходных и в ночное время, можно судить о самочувствии и моральном состоянии вашей команды.

Вы должны ограничить количество неотложных вызовов, особенно во внерабочее время: желательно, чтобы на инженера в неделю было не больше 1–2 внеурочных уведомлений. Если этот лимит регулярно превышается, вам следует откорректировать рассылку уведомлений, починить систему или нанять больше инженеров.

Просматривайте все неотложные вызовы хотя бы еженедельно и исправляйте/устраняйте любые ложные срабатывания или необязательные уведомления. Относитесь к этому серьезно, тогда и ваши коллеги будут серьезно относиться к уведомлениям. Если регулярно будить людей и вмешиваться в их частную жизнь своими необязательными вызовами, они начнут искать другую работу.

Инструменты и сервисы для работы с показателями

Далее перейдем к конкретике. Какие инструменты или сервисы следует использовать для сбора, анализа и передачи показателей? В пункте «Не нужно строить свою собственную инфраструктуру мониторинга» на с. 341 мы высказались о том, что для стандартной проблемы следует искать стандартное решение. Означает ли это, что вам обязательно нужно использовать сторонние, управляемые сервисы работы с показателями, такие как Datadog или New Relic?

Ответ на этот вопрос не такой очевидный. Эти сервисы предлагают множество мощных функций, однако могут быть достаточно дорогими, особенно в крупном масштабе. С точки зрения бизнеса не существует такой причины, по которой не стоило бы иметь собственный сервер показателей: ведь есть прекрасные бесплатные продукты с открытым исходным кодом.

Prometheus

Стандартным решением де-факто для работы с показателями в облачно-ориентированном мире является использование Prometheus. Эта система имеет очень широкое применение, особенно в связке с Kubernetes. С Prometheus может взаимодействовать почти все что угодно, поэтому при поиске систем мониторинга показателей данный вариант следует рассматривать в первую очередь.

Prometheus — это система мониторинга и набор инструментов для создания уведомлений с открытым исходным кодом на основе хронологических показателей. Главным компонентом Prometheus является сервер, который собирает и сохраняет показатели. Есть и другие, дополнительные компоненты, такие как инструмент для создания уведомлений (Alertmanager) и клиентские библиотеки для таких языков программирования, как Go, которыми вы можете оснастить свое приложение.

Все кажется довольно сложным, но на самом деле это очень простая система. Prometheus можно установить в кластер Kubernetes одной командой, используя стандартный чарт Helm (см. раздел «Helm: диспетчер пакетов для Kubernetes» на с. 102). После этого она начнет автоматически собирать показатели вашего кластера и любых приложений на ваш выбор.

Для извлечения показателей Prometheus устанавливает HTTP-соединение с вашим приложением на условленном порте и загружает все доступные данные. После сохранения в базе их можно запрашивать, визуализировать, создавать на их основе уведомления и т. д.

Метод сбора показателей, который использует Prometheus, называется активным мониторингом: сервер связывается с приложением и запрашивает данные. Противоположный, пассивный подход, используемый в других инструментах мониторинга наподобие StatsD, работает по-другому: приложение связывается с сервером, чтобы передать ему показатели.

Как и Kubernetes, проект Prometheus вдохновлен собственной инфраструктурой Google. Он был разработан компанией SoundCloud, но заимствовал много идей из инструмента под названием Borgmon. Последний, как можно догадаться по его

имени, был предназначен для мониторинга системы оркестрации контейнеров Google Borg (см. подраздел «От Borg до Kubernetes» на с. 39).

> Платформа Kubernetes опирается на десятилетний опыт компании Google в разработке собственной системы планирования кластеров Borg. Проект Prometheus почти не имеет отношения к Google, но во многом вдохновлен внутренней системой мониторинга этой компании под названием Borgmon, которая появилась примерно в то же время, что и Borg. В первом приближении можно сказать, что Kubernetes и Prometheus — это Borg и, соответственно, Borgmon для простых смертных. Обе эти системы являются «вторичными» и пытаются развивать самые удачные идеи своих прародителей, избегая их ошибок и тупиковых направлений.
>
> *Бьерн Рабенштейн. SoundCloud*[1]

Больше о проекте Prometheus можно почитать на его сайте (prometheus.io). Там же находятся и инструкции по установке и настройке в вашей среде.

Prometheus в основном занимается сбором и хранением показателей, а для визуализации, создания информационных панелей и уведомлений существуют другие высококачественные открытые инструменты. Например, Grafana (grafana.com) является мощной и функциональной системой визуализации хронологических данных (рис. 16.8).

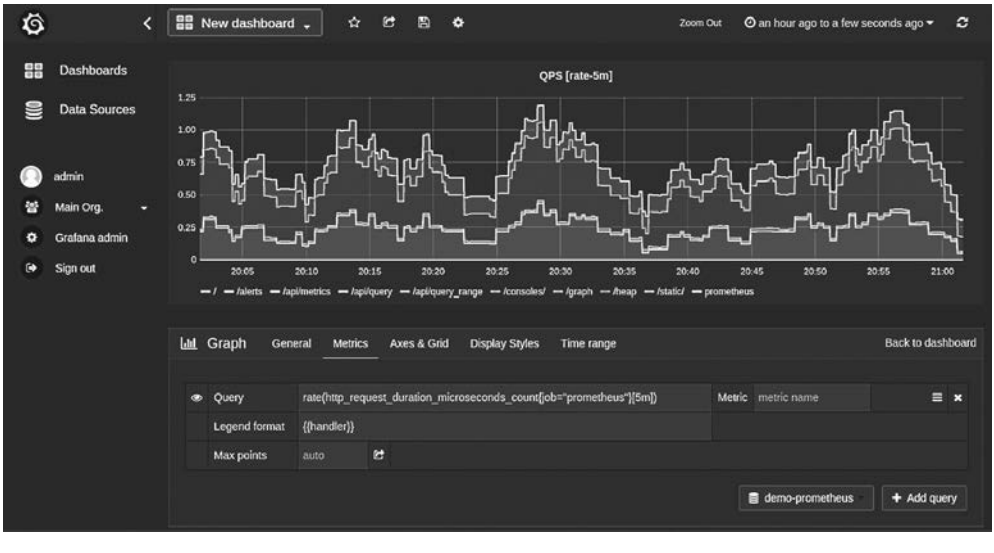

Рис. 16.8. Информационная панель Grafana с данными из Prometheus

[1] www.oreilly.com/radar/google-infrastructure-for-everyone-else.

В состав проекта Prometheus входит инструмент под названием Alertmanager (prometheus.io/docs/alerting/alertmanager), который может работать как в связке с основным продуктом, так и независимо от него. Задача Alertmanager состоит в том, чтобы получать уведомления из разных источников, включая серверы Prometheus, и обрабатывать их (см. раздел «Уведомления о показателях» на с. 366).

Первое, что делается с уведомлениями при их обработке, — это устраняются дубликаты. Затем Alertmanager группирует уведомления, между которыми обнаруживает какую-то связь. Например, серьезный сбой в работе сети может сгенерировать сотни отдельных уведомлений, но Alertmanager сформирует из них единое сообщение, чтобы не забрасывать дежурных вызовами.

Наконец, Alertmanager перенаправит обработанные уведомления в подходящий канал связи, такой как PagerDuty, Slack или электронная почта.

Формат показателей Prometheus поддерживается очень широким спектром инструментов и сервисов, что довольно удобно. Этот фактический стандарт является основой для OpenMetrics (openmetrics.io) — проекта Cloud Native Computing Foundation по созданию стандартного формата для данных показателей. Но вам не обязательно ждать, когда OpenMetrics станет реальностью, — в настоящее время почти любой сервис показателей, включая Stackdriver, Cloudwatch, Datadog и New Relic, умеет импортировать и понимать данные Prometheus.

Google Stackdriver

Проект Stackdriver был разработан в Google, однако поддерживает не только Google Cloud, но и AWS. Он может собирать показатели и журнальные записи из разнообразных источников, визуализировать и генерировать на их основе уведомления, а также автоматически обнаруживать и отслеживать ваши облачные ресурсы, включая виртуальные машины, базы данных и кластеры Kubernetes. Все эти данные Stackdriver делает доступными в центральной веб-консоли, в которой вы можете создавать собственные информационные панели и уведомления.

Stackdriver знает, как получить операционные показатели из таких популярных программных инструментов, как Apache, Nginx, Cassandra и Elasticsearch. Чтобы добавить показатели своего приложения, их можно экспортировать с помощью клиентской библиотеки Stackdriver.

Если вы работаете в Google Cloud, Stackdriver предоставляется бесплатно для всех показателей, относящихся к GCP, а отслеживание пользовательских данных или показателей из других облачных платформ тарифицируется помегабайтно и ежемесячно.

Stackdriver (cloud.google.com/monitoring) не может похвастаться гибкостью Prometheus или функциональностью более дорогих инструментов вроде Datadog, но это отличный способ начать работу с показателями без необходимости что-либо устанавливать или настраивать.

AWS Cloudwatch

У Amazon есть собственный продукт облачного мониторинга — Cloudwatch, который по своим возможностям похож на Stackdriver. Он интегрирован со всеми сервисами AWS и позволяет экспортировать пользовательские показатели с помощью Cloudwatch SDK или утилиты командной строки.

У Cloudwatch есть бесплатный тариф, позволяющий собирать основные показатели (такие как использование процессора для виртуальной машины) с пятиминутными интервалами и поддерживающий определенный набор информационных панелей, уведомлений и т. д. Все остальное является платным: тарифицируется каждый показатель, информационная панель и уведомление. Вы также можете платить за более детальные показатели (с одноминутными интервалами) для каждого сервера отдельно.

Как и Stackdriver, Cloudwatch (aws.amazon.com/cloudwatch) является простым, но эффективным инструментом. Если ваша основная облачная инфраструктура находится в AWS, Cloudwatch позволит легко начать работу с показателями, а для небольших развертываний этого в принципе может оказаться достаточно.

Azure Monitor

Monitor (docs.microsoft.com/en-us/azure/azure-monitor/overview) — это аналог Google Stackdriver и AWS Cloudwatch для Azure. Он собирает журнальные записи и показатели со всех ваших ресурсов, включая кластеры Kubernetes, позволяя их визуализировать и генерировать на их основе уведомления.

Datadog

Если сравнивать со встроенными инструментами от облачных провайдеров, такими как Stackdriver и Cloudwatch, Datadog является крайне сложной и мощной системой для мониторинга и анализа. Вам доступна интеграция более чем с 250 платформами и сервисами, включая все облачные сервисы от основных провайдеров и популярное ПО, такое как Jenkins, Varnish, Puppet, Consul и MySQL.

Datadog также предлагает компонент для мониторинга производительности приложений, с помощью которого вы сможете отслеживать и анализировать эффективность собственных программ. И неважно, что именно вы используете — Go, Java, Ruby или любую другую программную платформу: Datadog может собрать показатели, журнальные записи и трассировки из вашего ПО и ответить на следующие вопросы.

- Насколько тот или иной пользователь доволен моим сервисом?
- Какие десять клиентов получают самые медленные ответы от конкретной конечной точки?
- Какие из множества моих распределенных сервисов больше всего влияют на общую задержку выполнения запросов?

Помимо обычных информационных панелей (рис. 16.9) и уведомлений (с автоматизацией через Datadog API и клиентские библиотеки, включая Terraform), Datadog предоставляет такие возможности, как обнаружение аномалий на основе машинного обучения.

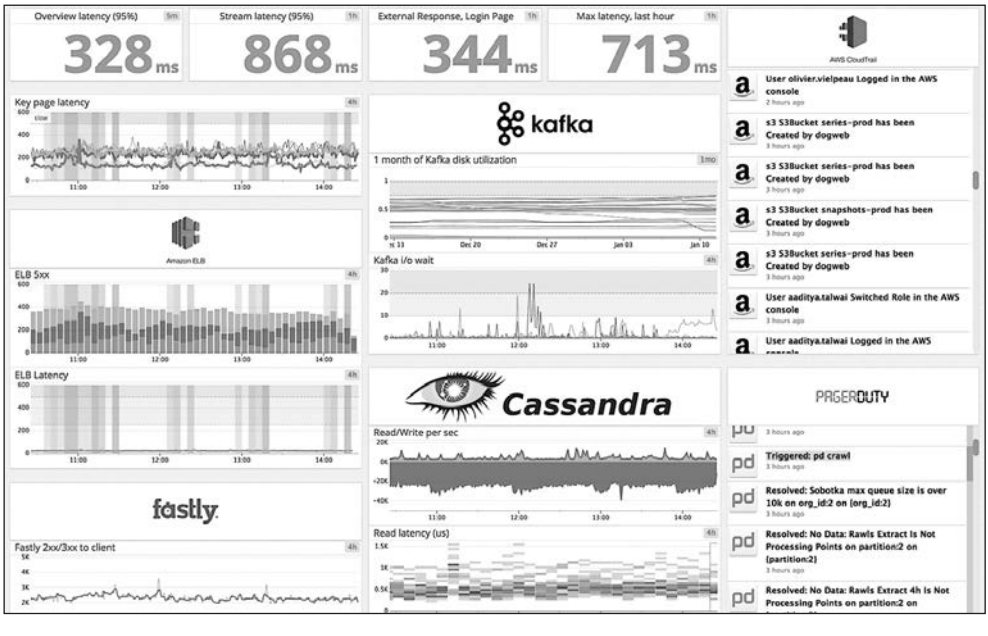

Рис. 16.9. Информационная панель Datadog

Как и следовало ожидать, Datadog — один из самых дорогих сервисов для мониторинга. Но если вы серьезно относитесь к наблюдаемости и у вас сложная

инфраструктура с приложениями, требующими высокой производительности, оно того стоит.

Больше о Datadog можно почитать на сайте проекта (www.datadoghq.com).

New Relic

New Relic — это давно устоявшаяся и широко используемая платформа для работы с показателями, предназначенная в основном для мониторинга производительности приложений. Ее главным достоинством является диагностика проблем с производительностью и узких мест внутри приложений и распределенных систем (рис. 16.10). Но она также предоставляет инфраструктурные показатели, мониторинг, уведомления, анализ ПО и любые другие возможности, какие вы только пожелаете.

Рис. 16.10. Раздел APM на информационной панели New Relic

Как и с любым сервисом уровня предприятия, мониторинг в New Relic стоит недешево, особенно в крупном масштабе. Если вы ищете высококачественную корпоративную платформу для работы с показателями, вам, скорее всего, подойдет либо New Relic (с большим акцентом на приложения), либо Datadog (с большим акцентом на инфраструктуру). Оба эти продукта предоставляют хорошую поддержку концепции *«инфраструктура как код»*. Например, и в New Relic и в Datadog для создания информационных панелей мониторинга и уведомлений можно использовать официальные провайдеры Terraform.

Резюме

«Семь раз отмерь, один раз отрежь» — любимая пословица многих инженеров. В облачно-ориентированном мире очень сложно понять, что происходит, не имея необходимых показателей и данных наблюдаемости. С другой стороны, избыток информации, как и ее нехватка, может сделать показатели бесполезными.

Хитрость в том, что вы изначально должны собирать те данные, которые нужно, затем обрабатывать их правильным образом для получения ответов на правильно поставленные вопросы и использовать для оповещения нужных людей в нужное время и о нужных событиях.

Главное, что вы должны запомнить из этой главы.

- Сосредотачивайтесь на ключевых показателях каждого сервиса: на запросах, ошибках и продолжительности (RED) — и каждого ресурса: на потреблении, степени загруженности и уровне ошибок (USE).
- Сделайте так, чтобы ваши приложения предоставляли пользовательские показатели. Это нужно как для внутренней наблюдаемости, так и для оценки бизнес-эффективности.
- На уровне кластера в число ключевых показателей Kubernetes входят количество узлов, среднее количество pod-оболочек на узле и потребление ресурсов узлами.
- На уровне развертывания отслеживайте объекты Deployment и реплики (особенно недоступные — они могут быть признаком нехватки емкости).
- На уровне контейнера отслеживайте использование ресурсов, состояние работоспособности/готовности, перезагрузки, сетевой трафик и сетевые ошибки.
- Создайте информационную панель для каждого сервиса со стандартной компоновкой и информационный излучатель, который будет сообщать показатели жизненно важных функций всей системы.

- Если человек получает уведомление, оно должно быть неотложным, важным и касаться проблемы, которую можно решить. Череда бесполезных уведомлений приводит к усталости сотрудников и сказывается на их моральном состоянии.
- Следите за тем, сколько неотложных вызовов получает ваша команда, особенно в ночное время и в выходные дни.
- Стандартом де-факто для работы с показателями в облачно-ориентированном мире является Prometheus, чей формат понимает почти любой другой инструмент.
- Среди сторонних управляемых сервисов для работы с показателями можно выделить Google Stackdriver, Amazon Cloudwatch, Datadog и New Relic.

Заключение

> Нет ничего сложнее и рискованнее и ничто не заставляет так усомниться в успехе, как попытка взять на себя ответственность за введение нового порядка.
>
> *Никколо Макиавелли*

Что ж, это было настоящее путешествие. Мы рассмотрели много тем, и все, что вошло в эту книгу, соответствовало одному простому принципу: мы считаем, что вам *нужно это знать*, чтобы использовать Kubernetes в промышленных условиях.

Говорят, что специалист — это тот, кто опережает вас в чтении руководства на одну страницу. Вполне возможно, что, читая нашу книгу, вы станете специалистом по Kubernetes в своей организации (по крайней мере, поначалу). Надеемся, что она будет для вас полезной. Но помните, что это лишь начало.

Что дальше?

Ниже перечислены ресурсы, которые могут вам пригодиться как в дальнейшем изучении Kubernetes и облачно-ориентированного мира, так и в том, чтобы быть в курсе последних новостей и разработок.

- slack.k8s.io. Официальный канал Kubernetes в Slack. Это хорошее место для того, чтобы что-то спросить и пообщаться с другими пользователями.
- discuss.kubernetes.io. Публичный форум для обсуждения всего, что связано с Kubernetes.
- kubernetespodcast.com. Еженедельный подкаст от Google. Эпизоды обычно длятся около 20 минут и содержат обзор новостей за прошедшую неделю и, как правило, интервью с кем-то, кто имеет отношение к Kubernetes.

- github.com/heptio/tgik. TGIK8s — еженедельный стрим, который проводит Джо Беда из Heptio. Видеотрансляция обычно продолжается около часа и содержит живую демонстрацию какого-то элемента экосистемы Kubernetes. Все видеозаписи сохраняются в архиве и доступны для свободного просмотра.

Не обойдем вниманием и блог, посвященный этой книге. Заглядывайте в него время от времени, чтобы узнать последние новости, обновления и статьи о книге.

Вот некоторые новостные рассылки, на которые вы, возможно, захотите подписаться: они посвящены таким темам, как разработка программного обеспечения, безопасность, DevOps и Kubernetes.

- KubeWeekly (twitter.com/kubeweekly) (от CNCF).
- SRE Weekly (sreweekly.com).
- DevOps Weekly (www.devopsweekly.com).
- DevOps'ish (devopsish.com).
- Security Newsletter (securitynewsletter.co).

Добро пожаловать на борт

Мы ничему не учимся, считая себя правыми.

Элизабет Бибеско

Вашим основным приоритетом при работе с Kubernetes должно быть стремление распространить свои знания как можно шире и при этом научиться как можно большему у других. Никто из нас не знает всего, но каждый знает что-то. И вместе мы можем во всем разобраться.

Не бойтесь экспериментировать. Создайте свое собственное демонстрационное приложение или возьмите наше и попытайтесь реализовать на нем то, что вам, скорее всего, понадобится в реальных условиях. Если все, что вы делаете, идеально работает, это означает, что вы недостаточно экспериментируете. Настоящие знания приходят с неудачами и попытками понять, что пошло не так и как это исправить. Чем чаще вы ошибаетесь, тем большему вы учитесь.

Всеми *своими* знаниями о Kubernetes мы обязаны собственным многочисленным неудачам. Желаем вам того же. Удачи!

Об авторах

Джон Арундел является консультантом с 30-летним опытом работы в компьютерной индустрии. Он написал несколько книг и работает со многими компаниями из разных стран, консультируя их в вопросах облачно-ориентированной инфраструктуры и Kubernetes. В свободное время увлекается серфингом, неплохо стреляет из пистолета и любительски играет на пианино. Живет в сказочном коттедже в Корнуолле, Англия.

Джастин Домингус — инженер системного администрирования, работающий в среде DevOps с Kubernetes и облачными технологиями. Ему нравится проводить время на свежем воздухе, пить кофе, ловить крабов и сидеть за компьютером. Живет в Сиэтле, штат Вашингтон, вместе с замечательным котом и еще более замечательной женой и по совместительству лучшим другом Эдриэнн.

Об обложке

На обложке этой книги изображен вознесенский фрегат (лат. *Fregata aquila*) — морская птица, которая водится только на острове Вознесения и на расположенном по соседству острове Боцмана в южной части Атлантического океана, где-то между Анголой и Бразилией. Сама птица и ее родной остров названы в честь христианского праздника Вознесения Господня, который пришелся на день, когда их впервые обнаружили.

С размахом крыльев более двух метров и весом менее 1,25 килограмма вознесенский фрегат непринужденно планирует над океаном и ловит рыбу, плавающую у поверхности (особенно летучую рыбу). Иногда он питается кальмарами и детенышами черепахи, а также тем, что удается украсть у других птиц. Его перья черные и блестящие, с зеленоватым и фиолетовым отливом. Самцов можно отличить по ярко-красному зобному мешку, который они надувают в поисках пары. Самки имеют чуть более тусклое оперение — с вкраплением коричневого и иногда белого у основания перьев. Как и другим птицам семейства фрегатовых, для них характерны крючковатый клюв, раздвоенный хвост и остроконечные крылья.

Вознесенский фрегат размножается на скалистых выступах своей островной среды обитания. Вместо того чтобы вить гнездо, он делает в земле углубление и покрывает его сверху перьями, булыжниками и костями. Самка откладывает единственное яйцо, а за птенцом ухаживают оба родителя на протяжении 6–7 месяцев, пока тот наконец не научится летать. Из-за низкой частоты успешных спариваний и ограниченной среды обитания этот вид обычно относят к уязвимым.

Первые поселенцы на острове Вознесения появились в начале XIX века — это были британцы, разместившиеся там в военных целях. В наши дни на острове находятся станции слежения NASA и Европейского космического агентства, ретранслятор BBC World Service и антенна GPS, одна из четырех во всем мире. На протяжении большей части XIX и XX столетий фрегатовые птицы могли

размножаться только на небольшом скалистом островке Боцмана у побережья острова Вознесения, поскольку дикие кошки убивали их птенцов. В 2002 году Королевское общество защиты птиц начало кампанию по избавлению острова от кошек, и уже несколько лет спустя фрегатовые птицы снова стали гнездиться на острове Вознесения.

Многие животные с обложек издательства O'Reilly находятся под угрозой исчезновения — все они важны для нашей планеты. О том, как им помочь, можно узнать на сайте animals.oreilly.com.

Джон Арундел, Джастин Домингус

Kubernetes для DevOps: развертывание, запуск и масштабирование в облаке

Перевел с английского Р. Волошко

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Кудрейко*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Е. Рафалюк-Бузовская*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 24.01.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 30,960. Тираж 700. Заказ 0000.