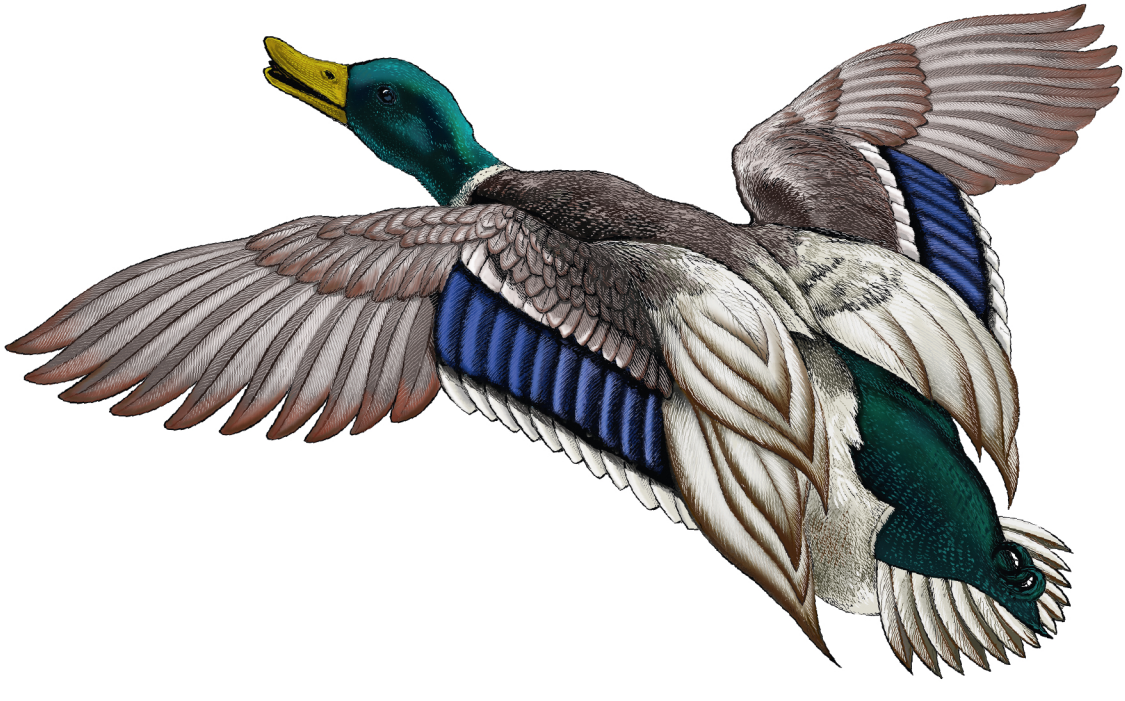
O'REILLY®

Kubernetes: лучшие практики

Раскрой потенциал главного инструмента в отрасли



Брендан Бернс, Эдди Вильяльба
Дейв Штребель, Лахлан Эвенсон

Kubernetes Best Practices

Blueprints for Building Successful Applications on Kubernetes

Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Kubernetes: лучшие практики

Раскрой потенциал главного инструмента в отрасли

Брендан Бернс, Эдди Вильяльба
Дейв Штребель, Лахлан Эвенсон

ПИТЕР®

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

2021

ББК 32.973.2-018+32.988.02
УДК 004.4+004.7
К99

Бернс Брендан, Вильяльба Эдди, Штребель Дейв, Эвенсон Лахлан

К99 Kubernetes: лучшие практики. — СПб.: Питер, 2021. — 288 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1688-1

Положитесь на опыт профессионалов, успешно применяющих и развивающих проект Kubernetes. Инженеры Microsoft предлагают лучшие приемы оркестрации контейнеров. Их практики сложились в процессе разработки распределенных систем, на ответственных и нагруженных проектах. Вам останется лишь слегка адаптировать код.

Книга идеально подойдет тем, кто уже знаком с Kubernetes, но еще не умеет использовать его максимально эффективно. Вы узнаете все, что необходимо для создания классного Kubernetes-приложения, в том числе:

- Подготовка окружения и разработка приложений в Kubernetes.
- Паттерны мониторинга и защиты ваших систем, управления обновлениями.
- Сетевые политики Kubernetes и роли сервисных сетей в экосистеме.
- Использование Kubernetes в задачах машинного обучения.

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

ББК 32.973.2-018+32.988.02
УДК 004.4+004.7

ISBN 978-1492056478 англ.

Authorized Russian translation of the English edition of Kubernetes Best Practices: ISBN 9781492056478 © 2020 Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1688-1

© Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке, оформление ООО Издательство «Питер», 2021
© Серия «Для профессионалов», 2021
© Анатолий Павлов, пер. с англ., 2020

Краткое содержание

Оглавление

Введение

Кому стоит прочесть эту книгу

Kubernetes — фактический стандарт для облачно-ориентированной разработки. Это эффективный инструмент, который может упростить создание ваших приложений, ускорить развертывание и сделать его более надежным. Но для раскрытия всего потенциала этой платформы нужно научиться ее корректно использовать. Книга предназначена для всех, кто развертывает реальные приложения в Kubernetes и заинтересован в изучении паттернов проектирования и методик, применимых к этим приложениям.

Важно понимать: это не введение в Kubernetes. Мы исходим из того, что вы уже имеете общее представление об API и инструментарии данной платформы и знаете, как создавать и администрировать кластеры на ее основе. Познакомиться с Kubernetes можно, в частности, прочитав книгу *Kubernetes: Up and Running* (O'Reilly) (https://oreil.ly/ziNRK).

Эта книга предназначена для читателей, желающих подробно изучить процесс развертывания конкретных приложений в Kubernetes. Она будет полезной как тем, кто собирается развернуть в Kubernetes свое первое приложение, так и специалистам с многолетним опытом использования данной платформы.

Почему мы написали эту книгу

Мы, четыре автора этой книги, многократно помогали развертывать приложения в Kubernetes. Мы видели, какие трудности возникали у разных пользователей, и помогали решать их. Приступая к написанию книги, мы хотели поделиться своим опытом, чтобы еще больше людей могло извлечь пользу из усвоенных нами уроков. Надеемся, таким образом нам удастся сделать наши знания общедоступными и благодаря этому вы сможете самостоятельно развертывать и администрировать свои приложения в Kubernetes.

Структура книги

Эту книгу можно читать последовательно, от первой до последней страницы, однако на самом деле мы задумывали ее как набор независимых глав. В каждой главе дается полноценный обзор определенной задачи, выполнимой с помощью Kubernetes. Мы ожидаем, что вы углубитесь в материал, чтобы узнать о конкретной проблеме или интересующем вас вопросе, а затем будете периодически обращаться к книге при возникновении новых запросов.

Несмотря на столь четкое разделение, некоторые темы будут встречаться вам на протяжении всей книги. Разработке приложений в Kubernetes посвящено сразу несколько глав. Глава 2 описывает рабочий процесс. В главе 5 обсуждаются непрерывная интеграция и тестирование. В главе 15 речь идет о построении на основе Kubernetes высокоуровневых платформ, а в главе 16 рассматривается управление stateless- и stateful-приложениями. Управлению сервисами в Kubernetes также отводится несколько глав. Глава 1 посвящена подготовке простого сервиса, а глава 3 — мониторингу и метрикам. В главе 4 вы научитесь управлять конфигурацией, а в главе 6 — версиями и релизами. В главе 7 описывается процесс глобального развертывания приложения.

Другая обширная тема — работа с кластерами. Сюда относятся управление ресурсами (глава 8), сетевые возможности (глава 9), безопасность pod (глава 10), политики и управляемость (глава 11), управление несколькими кластерами (глава 12), а также контроль доступа и авторизацию (глава 17). Кроме того, есть полностью самостоятельные главы, посвященные машинному обучению (глава 14) и интеграции с внешними сервисами (глава 13).

Прежде чем браться за выполнение реальной задачи, не помешает прочесть соответствующие главы, хотя мы надеемся, что вы будете использовать эту книгу как справочник.

Условные обозначения

В этой книге используются следующие условные обозначения:

Курсив

Курсивом выделены новые термины и важные слова.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных,

типы данных, переменные среды, операторы и ключевые слова, имена файлов и их расширения.

`Моноширинный жирный шрифт`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

`Моноширинный курсивный шрифт`

Показывает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок, каталогов.

Этот рисунок указывает на совет или предложение.

Такой рисунок указывает на общее замечание.

Этот рисунок указывает на предупреждение.

Использование примеров кода

Дополнительный материал (примеры кода, упражнения и т. д.) доступен для скачивания по адресу oreil.ly/KBPsample.

Если при использовании примеров кода у вас возникнут технические вопросы или проблемы, то, пожалуйста, обращайтесь к нам по адресу bookquestions@oreilly.com.

Назначение книги — помочь решить ваши задачи. В ваших программах и документации разрешается использовать предложенный пример кода. Не нужно связываться с нами, если вы не воспроизводите его существенную часть: например, когда включаете в свою программу несколько фрагментов кода,

приведенного здесь. Однако продажа или распространение компакт-дисков с примерами из книг издательства O'Reilly требует отдельного разрешения. Вы можете свободно цитировать эту книгу с примерами кода, отвечая на вопрос, но если хотите включить существенную часть приведенного здесь кода в документацию своего продукта, то вам следует связаться с нами.

Мы приветствуем, но не требуем отсылки на оригинал. Отсылка обычно состоит из названия, имени автора, издательства, ISBN и копирайта. Например, «Kubernetes: лучшие практики», Брендан Бернс, Эдди Вильяльба, Дейв Штребель, Лахлан Эвенсон (Питер). Copyright 2020 Brendan Burns, Eddie Villalba, Dave Strebel and Lachlan Evenson. 978-5-4461-1688-1.

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

Благодарности

Брендан хотел бы поблагодарить свою чудесную семью: Робин, Джулию и Итана — за любовь и поддержку всех его действий; сообщество Kubernetes, благодаря которому все это стало возможным; своих потрясающих соавторов — без них книга не появилась бы на свет.

Дейв благодарит за поддержку свою прекрасную жену Джен и трех детей: Макса, Мэдди и Мэйсона. Он также хотел бы поблагодарить сообщество Kubernetes за все советы и помощь, полученные на протяжении многих лет. И наконец, он хотел бы выразить признательность своим соавторам, которые помогли воплотить этот проект в жизнь.

Лахлан хотел бы поблагодарить свою жену и троих детей за их любовь и поддержку. Он также хотел бы сказать спасибо всем участникам сообщества Kubernetes, в том числе замечательным людям, на протяжении этих лет находившим время, чтобы поделиться с ним знаниями. Он хотел бы выразить особую признательность Джозефу Сандовалу за наставничество. И наконец, Лахлан не может не поблагодарить своих фантастических соавторов, благодаря которым появилась эта книга.

Эдди хотел бы поблагодарить свою жену Сандру за моральную поддержку и возможность заниматься этим изданием часами напролет, в то время как она сама была в последнем триместре первой беременности. Он также хотел бы сказать спасибо своей дочери Джованне за дополнительную моти-

вацию. В завершение Эдди хотел бы выразить признательность сообществу Kubernetes и своим соавторам, на которых он всегда равнялся при работе с облачно-ориентированными технологиями.

Мы хотели бы поблагодарить Вирджинию Уилсон за ее помощь в работе над рукописью и объединении всех наших идей, а также Бриджет Кромхаут, Билгина Ибряма, Роланда Хуса и Джастина Домингуса за их внимание к деталям.

От издательства

Ваши замечания, предложения, вопросы отправляйте по электронному 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.

ГЛАВА 1

Создание простого сервиса

В этой главе описываются приемы создания простого многоуровневого приложения в Kubernetes, представляющего собой веб-сервис и базу данных. Это далеко не самый сложный пример, однако он послужит хорошей отправной точкой, на которую следует ориентироваться при управлении приложениями в Kubernetes.

Обзор приложения

Приложение, которое мы будем использовать в данном примере, не отличается особой сложностью. Это простой журнальный сервис, хранящий свои данные в Redis. Он также содержит отдельный сервер статических файлов на основе NGINX и предоставляет единый URL с двумя веб-путями. Первый путь, https://my-host.io, предназначен для файлового сервера, а второй, https://my-host.io/api, — для программного интерфейса приложения (application programming interface, API) в формате REST. Мы будем использовать SSL-сертификаты от Let's Encrypt (https://letsencrypt.org). На рис. 1.1 показана схема приложения. Для его построения мы воспользуемся сначала конфигурационными файлами YAML, а затем Heml-чартами.

Управление конфигурационными файлами

Прежде чем погружаться в подробности развертывания данного приложения в Kubernetes, следует поговорить о том, как управлять конфигурацией. В Kubernetes все представлено в *декларативном* виде. Это значит, что для определения всех аспектов своего приложения вы сначала описываете, каким должно быть его состояние в кластере (обычно в формате YAML или JSON). Декларативный подход куда более предпочтителен по сравнению с *императивным*, в котором состояние кластера представляет собой совокупность внесенных в него изменений. В случае императивной конфигурации очень сложно понять и воспроизвести состояние, в котором находится

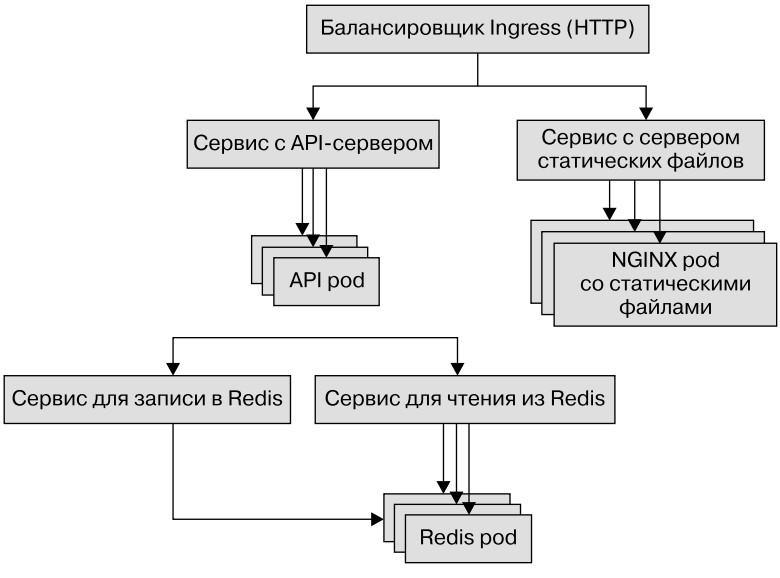


Рис. 1.1. Схема приложения

кластер. Это существенно затрудняет диагностику и исправление проблем, возникающих в приложении.

Предпочтительный формат для объявления состояния приложения — YAML, хотя Kubernetes поддерживает и JSON. Дело в том, что YAML выглядит несколько более компактно и лучше подходит для редактирования вручную. Однако стоит отметить: этот формат чувствителен к отступам, из-за чего многие ошибки в конфигурационных файлах связаны с некорректными пробельными символами. Если что-то идет не так, то имеет смысл проверить отступы.

Поскольку декларативное состояние, хранящееся в этих YAML-файлах, служит источником истины, из которого ваше приложение черпает информацию, правильная работа с состоянием — залог успеха. В ходе его редактирования вы должны иметь возможность управлять изменениями, проверять их корректность, проводить аудит их авторов и откатывать изменения в случае неполадок. К счастью, в контексте разработки ПО для всего этого уже есть подходящие инструменты. В частности, практические рекомендации, относящиеся к управлению версиями и аудиту изменений кода, можно смело использовать в работе с декларативным состоянием приложения.

В наши дни большинство людей хранят конфигурацию для Kubernetes в Git. И хотя выбор той или иной системы контроля версий непринципиален, многие инструменты в экосистеме Kubernetes рассчитаны на хранение файлов в Git-репозитории. В сфере аудита изменения кода все не так однозначно: многие используют локальные инструменты и сервисы, хотя платформа GitHub, несомненно, довольно популярна. Независимо от того, как вы реализуете процесс аудита изменения кода для конфигурации своего приложения, относиться к нему следует с тем же вниманием, что и к системе контроля версий.

Для организации компонентов приложения обычно стоит использовать структуру папок файлов системы. *Сервис приложения* (какой бы смысл ни вкладывала ваша команда в это понятие) обычно хранится в отдельном каталоге, а его компоненты — в подкаталогах.

В этом примере мы структурируем файлы таким образом:

```
journal/
  frontend/
  redis/
  fileserver/
```

Внутри каждого каталога находятся конкретные YAML-файлы, необходимые для определения сервиса. Как вы позже сами увидите, по мере развертывания нашего приложения в разных регионах или кластерах эта структура каталогов будет все более усложняться.

Создание реплицированного сервиса с помощью ресурса Deployment

Мы начнем описание нашего приложения с клиентской части и будем продвигаться вниз. В качестве этой части журнала будет выступать приложение для Node.js, написанное на языке TypeScript. Его код (https://oreil.ly/70kFT) слишком велик, чтобы приводить его здесь целиком. На порте 8080 работает HTTP-сервис, который обслуживает запросы к `/api/*` и использует сервер Redis для добавления, удаления и вывода актуальных записей журнала. Вы можете собрать это приложение в виде образа контейнера, используя включенный в его код файл Dockerfile, и загрузить его в собственный репозиторий образов. Затем вы сможете подставить его имя в YAML-файлы, приведенные ниже.

Практические рекомендации по управлению образами

В целом сборка и обслуживание образов контейнеров выходит за рамки этой книги, но все же будет уместно перечислить некоторые рекомендации. Процесс сборки образов как таковой уязвим к «атакам на поставщиков». В ходе подобных атак злоумышленник внедряет код или двоичные файлы в одну из зависимостей, которая хранится в доверенном источнике и участвует в сборке вашего приложения. Поскольку это создает слишком высокий риск, в сборке должны участвовать только хорошо известные и доверенные провайдеры образов. В качестве альтернативы все образы можно собирать с нуля; для некоторых языков (например, Go) это не составляет труда, поскольку они могут создавать статические исполняемые файлы, но в интерпретируемых языках, таких как Python, JavaScript и Ruby, это может быть большой проблемой.

Некоторые рекомендации касаются выбора имен для образов. Теоретически тег с версией образа контейнера в реестре можно изменить, но вы никогда не должны этого делать. Хороший пример системы именования — сочетание семантической версии и SHA-хеша фиксации кода, из которой собирается образ (например, `v1.0.1-bfeda01f`). Если версию не указать, то по умолчанию используется значение `latest`. Оно может быть удобно в процессе разработки, но в промышленных условиях данного значения лучше избегать, так как оно явно изменяется при создании каждого нового образа.

Создание реплицированного приложения

Наше клиентское приложение является *stateless* (не хранит свое состояние), делегируя данную функцию серверу Redis. Благодаря этому его можно реплицировать произвольным образом без воздействия на трафик. И хотя наш пример вряд ли будет испытывать серьезные нагрузки, все же неплохо использовать как минимум две реплики (копии): это позволяет справляться с неожиданными сбоями и выкатывать новые версии без простоя.

В Kubernetes есть объект `ReplicaSet`, отвечающий за репликацию контейнеризованных приложений, но его лучше не использовать напрямую. Для наших задач подойдет объект `Deployment`, который сочетает в себе возможности объекта `ReplicaSet`, систему управления версиями и поддержку поэтапного развертывания обновлений. Объект `Deployment` позволяет применять встроенные в Kubernetes механизмы для перехода от одной версии к другой.

Ресурс `Deployment` нашего приложения выглядит следующим образом:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: frontend
  name: frontend
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - image: my-repo/journal-server:v1-abcde
        imagePullPolicy: IfNotPresent
        name: frontend
        resources:
          requests:
            cpu: "1.0"
            memory: "1G"
          limits:
            cpu: "1.0"
            memory: "1G"
```

В этом ресурсе следует обратить внимание на несколько деталей. Так, для идентификации экземпляров `ReplicaSet`, объекта `Deployment` и создаваемых им pod используются метки (`labels`). Мы добавили метку `layer: frontend` для всех этих объектов, благодаря чему они теперь находятся в общем слое (`layer`) и их можно просматривать вместе с помощью одного запроса. Как вы увидите позже, мы будем применять данный подход и при добавлении других ресурсов.

Помимо этого, мы добавили комментарии в разные участки YAML-файла. Как и комментарии в программном коде, они не попадут в итоговый ресурс, хранящийся на сервере; их задача — сделать конфигурацию более понятной для тех, кто не видел ее раньше.

Обратите внимание и на то, что для контейнеров в ресурсе `Deployment` установлены запросы ресурсов `Request` и `Limit` с одинаковыми значениями.

`Request` гарантирует выделение определенного объема ресурсов на сервере, на котором запущено приложение. `Limit` — максимальное потребление ресурсов, разрешенное к использованию контейнером. На первых порах установка одинаковых значений для этих двух запросов обеспечивает наиболее предсказуемое поведение приложения. Но за это приходится платить неоптимальным использованием ресурсов. С одной стороны, когда `Request` и `Limit` равны, ваше приложение не тратит слишком много процессорного времени и не потребляет лишние ресурсы при бездействии. С другой — если не проявить крайнюю осторожность при подборе этих значений, то вы не можете в полной мере использовать ресурсы сервера. Начав лучше ориентироваться в модели управления ресурсами Kubernetes, вы сможете попробовать откорректировать параметры `Request` и `Limit` своего приложения по отдельности, но в целом большинство пользователей предпочитают пожертвовать эффективностью в угоду стабильности, получаемой в результате предсказуемости.

Итак, мы определили ресурс `Deployment`. Теперь сохраним его в системе контроля версий и развернем в Kubernetes:

```
git add frontend/deployment.yaml
git commit -m "Added deployment" frontend/deployment.yaml
kubectl apply -f frontend/deployment.yaml
```

Рекомендуется также следить за тем, чтобы содержимое вашего кластера в точности соответствовало содержимому репозитория. Для этого лучше всего использовать методику GitOps и брать код для промышленной среды только из определенной ветки системы контроля версий. Данный процесс можно автоматизировать с помощью непрерывной интеграции (Continuous Integration, CI) и непрерывной доставки (Continuous Delivery, CD). Это позволяет гарантировать соответствие между репозиторием и промышленной системой. Для простого приложения полноценный процесс CI/CD может показаться избыточным, но автоматизация как таковая, даже если не брать во внимание повышение надежности, которое она обеспечивает, обычно стоит затраченных усилий. Внедрение CI/CD в уже существующий проект с императивным развертыванием — чрезвычайно сложная задача.

В следующих разделах мы обсудим другие фрагменты этого YAML-файла (например, `ConfigMap` и секретные тома) и качество обслуживания (Quality of Service) pod-оболочки.

Настройка внешнего доступа для HTTP-трафика

Контейнеры нашего приложения уже развернуты, но к нему все еще нельзя обратиться. Ресурсы кластера по умолчанию недоступны снаружи. Чтобы с ними мог работать кто угодно, нам нужно создать объект `Service` и балансировщик нагрузки; это позволит присвоить контейнерам внешний IP-адрес и направить к ним трафик. Для этого мы воспользуемся двумя ресурсами. Первый — это `Service`, который распределяет (балансирует) трафик, поступающий по TCP или UDP. В нашем примере мы задействуем протокол TCP. Второй ресурс — объект `Ingress`, обеспечивающий балансировку нагрузки с гибкой маршрутизацией запросов в зависимости от доменных имен и HTTP-путей. Вам, наверное, интересно, зачем такому простому приложению, как наше, настолько сложный ресурс, коим является `Ingress`. Но в последующих главах вы увидите, что даже в этом незамысловатом примере обслуживаются HTTP-запросы из двух разных сервисов. Более того, наличие `Ingress` на границе кластера обеспечивает гибкость, необходимую для дальнейшего расширения нашего сервиса.

Прежде чем определять ресурс `Ingress`, следует создать Kubernetes `Service`, на который он будет указывать. А чтобы связать этот `Service` с pod, созданными в предыдущем разделе, мы воспользуемся метками. Определение `Service` выглядит намного проще, чем ресурс `Deployment`:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: frontend
  name: frontend
  namespace: default
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: frontend
  type: ClusterIP
```

Теперь можно определить ресурс `Ingress`. Он, в отличие от `Service`, требует наличия в кластере контейнера с подходящим контроллером. Контроллеры бывают разные: одни из них предоставляются облачными провайдерами, а другие основываются на серверах с открытым исходным кодом. Если вы выбрали открытую реализацию `Ingress`, то для ее установки и обслужива-

ния лучше использовать диспетчер пакетов Helm (helm.sh). Популярностью пользуются такие реализации, как `nginx` и `haproxy`:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: frontend-ingress
spec:
  rules:
  - http:
      paths:
      - path: /api
        backend:
          serviceName: frontend
          servicePort: 8080
```

Конфигурация приложения с помощью ConfigMap

Любое приложение должно быть в той или иной степени настраиваемым. Это может касаться количества отображаемых журнальных записей на странице, цвета фона, специального праздничного представления и многих других параметров. Обычно такую конфигурационную информацию лучше всего хранить отдельно от самого приложения.

У такого разделения есть несколько причин. Прежде всего, у вас может быть несколько одинаковых исполняемых файлов с разными настройками, зависящими от определенных условий. Например, вы можете предусмотреть две отдельные страницы, где будут поздравления с Пасхой в Европе и китайским Новым годом в Китае. Помимо региональной специфики, разделение вызвано еще одной причиной — гибкостью. Двоичные релизы, как правило, содержат некоторый новый функционал; если включать их непосредственно в коде, то это потребует выпуска нового двоичного файла, что может быть затратным и медленным процессом.

Конфигурация позволяет быстро (и даже динамически) активизировать и деактивизировать возможности в зависимости от потребностей пользователей или программных сбоев. Вы можете выкатывать и откатывать каждую отдельную функцию. Такая гибкость позволяет непрерывно развивать приложение, даже если некоторые из его возможностей приходится выключать из-за плохой производительности или некорректного поведения.

В Kubernetes такого рода конфигурация представлена ресурсом под названием `ConfigMap`. Данный ресурс содержит пары типа «ключ — значение»,

описывающие конфигурационную информацию или файл. Эта информация предоставляется pod с помощью файлов или переменных среды. Представьте, что вам нужно сделать настраиваемым количество журнальных записей, отображаемых на каждой странице. Для этого можно создать ресурс `ConfigMap`:

```
kubectl create configmap frontend-config --from-literal=journalEntries=10
```

Затем вы должны предоставить конфигурационную информацию в виде переменной среды в самом приложении. Для этого в раздел `containers` ресурса `Deployment`, который вы определили ранее, можно добавить следующий код:

```
...
# Массив контейнеров в PodTemplate внутри Deployment.
containers:
  - name: frontend
    ...
    env:
    - name: JOURNAL_ENTRIES
      valueFrom:
        configMapKeyRef:
          name: frontend-config
          key: journalEntries
...
```

Это один из примеров использования `ConfigMap` для конфигурации приложения, но в реальных условиях изменения в данный ресурс можно вносить на регулярной основе: еженедельно или еще чаще. У вас может возникнуть соблазн делать это напрямую, редактируя сам файл `ConfigMap`, но это не самый удачный подход. Тому есть несколько причин: прежде всего, изменение конфигурации само по себе не инициирует обновление существующих pod. Чтобы применить настройки, pod нужно перезапустить. В связи с этим развертывание не зависит от работоспособности приложения и может происходить по мере необходимости или произвольным образом.

Вместо этого номер версии лучше указывать и в имени самого ресурса `ConfigMap`. Например, `frontend-config-v1`, а не `frontend-config`. Если вам нужно внести изменение, то не обновляйте существующую конфигурацию, а создайте новый экземпляр `ConfigMap` с версией `v2` и затем отредактируйте `Deployment` так, чтобы он его использовал. Благодаря этому развертывание происходит автоматически, с использованием соответствующих проверок работоспособности и пауз между изменениями. Более того, если вам нужно откатить обновление, то конфигурация `v1` по-прежнему находится в кластере и, чтобы переключиться на нее, достаточно еще раз отредактировать `Deployment`.

Управление аутентификацией с помощью объектов Secret

До сих пор мы обходили вниманием сервис Redis, к которому подключена клиентская часть нашего приложения. Но в любом реальном проекте соединение между сервисами должно быть защищенным. Это отчасти делается в целях повышения безопасности пользователей и их данных, но в то же время необходимо для предотвращения ошибок, таких как подключение клиентской части к промышленной базе данных.

Для аутентификации в сервере Redis используется обычный пароль, который было бы удобно хранить в исходном коде приложения или в одном из файлов вашего образа. Однако оба варианта являются плохими по целому ряду причин. Для начала это раскрывает ваши секретные данные (пароль) в среде, в которой может не быть никакой системы контроля доступа. Если пароль находится в исходном коде, то доступ к вашему репозиторию эквивалентен доступу ко всем секретным данным. Это, скорее всего, плохое решение. Обычно доступ к исходному коду имеет более широкий круг пользователей, чем к серверу Redis. Точно так же не всех пользователей, имеющих доступ к образу контейнера, следует допускать к промышленной базе данных.

Помимо проблем с контролем доступа, есть еще одна причина, почему не стоит привязывать секретные данные к исходному коду и/или образам: параметризация. Вы должны иметь возможность использовать одни и те же образы и код в разных средах (отладочной, канареечной или промышленной). Если секретные данные привязаны к исходному коду или образу, то вам придется собирать новый образ (или код) для каждой отдельной среды.

В прошлом разделе вы познакомились с ресурсом `ConfigMap` и, наверное, думаете, что пароль можно было бы хранить в качестве конфигурации и затем передавать его приложению. Действительно, конфигурация и секретные данные отделяются от приложения по тому же принципу. Но дело в том, что последние сами по себе являются важной концепцией. Вам вряд ли захочется управлять контролем доступа, администрированием и обновлением секретных данных так, как вы это делаете с конфигурацией. Что еще важнее, ваши разработчики должны по-разному *воспринимать* доступ к конфигурации и секретным данным. В связи с этим Kubernetes располагает встроенным ресурсом `Secret`, предназначенным специально для этих целей.

Вы можете создать секретный пароль для своей базы данных Redis следующим образом:

```
kubectl create secret generic redis-passwd --from-literal=passwd=${RANDOM}
```

Очевидно, что в качестве пароля лучше не использовать просто случайные числа. К тому же вам, вероятно, захочется подключить сервис для управления секретными данными или ключами; это может быть решение вашего облачного провайдера, такое как Microsoft Azure Key Vault, или открытый проект наподобие HashiCorp Vault. Обычно сервисы по управлению ключами имеют более тесную интеграцию с ресурсами `Secret` в Kubernetes.

По умолчанию секретные данные в Kubernetes хранятся в незашифрованном виде. Если вам нужно шифрование, то можете воспользоваться интеграцией с провайдером ключей; вы получите ключ, с помощью которого будут шифроваться все секретные данные в кластере. Это защищает от непосредственных атак на базу данных etcd, но вам все равно необходимо позаботиться о безопасном доступе через API-сервер Kubernetes.

Сохранив пароль к Redis в виде объекта `Secret`, вы должны *привязать* его к приложению, которое развертывается в Kubernetes. Для этого можно использовать ресурс `Volume` (том). Том — это, в сущности, файл или каталог с возможностью подключения к запущенному контейнеру по заданному пользователем пути. В случае с секретными данными том создается в виде файловой системы tmpfs, размещенной в оперативной памяти, и затем подключается к контейнеру. Благодаря этому, даже если злоумышленник имеет физический доступ к серверу (что маловероятно в облаке, но может случиться в вычислительном центре), ему будет намного сложнее заполучить секретные данные.

Чтобы добавить секретный том в объект `Deployment`, вам нужно указать в YAML-файле последнего два дополнительных раздела. Первый раздел, `volumes`, добавляет том в pod:

```
...
  volumes:
  - name: passwd-volume
    secret:
    secretName: redis-passwd
```

Затем том нужно подключить к определенному контейнеру. Для этого в описании контейнера следует указать поле `volumeMounts`:

```
...
  volumeMounts:
  - name: passwd-volume
    readOnly: true
    mountPath: "/etc/redis-passwd"
...
```

Благодаря этому том становится доступным для клиентского кода в каталоге redis-passwd. Итоговый объект `Deployment` будет выглядеть следующим образом:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: frontend
  name: frontend
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - image: my-repo/journal-server:v1-abcde
        imagePullPolicy: IfNotPresent
        name: frontend
        volumeMounts:
        - name: passwd-volume
          readOnly: true
          mountPath: "/etc/redis-passwd"
        resources:
          requests:
            cpu: "1.0"
            memory: "1G"
          limits:
            cpu: "1.0"
            memory: "1G"
      volumes:
        - name: passwd-volume
          secret:
            secretName: redis-passwd
```

Благодаря этой конфигурации наше клиентское приложение имеет доступ к паролю, который позволяет ему войти в сервис Redis. Аналогичным образом использование пароля настраивается и в самом сервисе; мы подключаем секретный том к Redis pod и загружаем пароль из файла.

Stateful-развертывание простой базы данных

Развертывание stateful принципиально не отличается от развертывания клиентского приложения, которое мы рассматривали в предыдущих разделах, однако наличие состояния вносит дополнительные сложности. Прежде всего, планирование функционирования pod в Kubernetes зависит от ряда факторов, таких как работоспособность узла, обновление или перебалансировка. Если данные экземпляра Redis хранятся на каком-то конкретном сервере или в самом контейнере, то будут потеряны при миграции или перезапуске данного контейнера. Чтобы этого избежать, при выполнении в Kubernetes stateful-приложений нужно обязательно использовать удаленные *постоянные тома* (`PersistentVolumes`).

Kubernetes поддерживает различные реализации объекта `PersistentVolume`, но все они имеют общие свойства. Как и секретные тома, описанные ранее, они привязываются к pod и подключаются к контейнеру по определенному пути. Их особенностью является то, что они обычно представляют собой удаленные хранилища, которые подключаются по некоему сетевому протоколу: или файловому (как в случае с NFS и SMB), или блочному (как в случае с iSCSI, облачными дисками и т. д.). В целом для таких приложений, как базы данных, предпочтительны блочные диски, поскольку обеспечивают лучшую производительность. Но если скорость работы не настолько важна, то файловые диски могут быть более гибкими.

Управление состоянием, как в Kubernetes, так и в целом, — сложная задача. Если среда, в которой вы работаете, поддерживает сервисы с сохранением состояния (stateful) (например, MySQL или Redis), то обычно лучше использовать именно их. Сначала тарифы на SaaS (Software as a Service — программное обеспечение как услуга) могут показаться высокими, но если учесть все операционные требования к поддержанию состояния (резервное копирование, обеспечение локальности и избыточности данных и т. д.) и тот факт, что наличие состояния осложняет перемещение приложений между кластерами Kubernetes, то становится

очевидно, что в большинстве случаев высокая цена SaaS себя оправдывает. В средах с локальным размещением, где сервисы SaaS недоступны, имеет смысл организовать отдельную команду специалистов, которая будет предоставлять услугу хранения данных в рамках всей организации. Это, несомненно, лучше, чем позволять каждой команде выкатывать собственное решение.

Для развертывания сервиса Redis мы воспользуемся ресурсом `StatefulSet`. Это дополнение к `ReplicaSet`, которое появилось уже после выхода первой версии Kubernetes и предоставляет более строгие гарантии, такие как согласованные имена (никаких случайных хешей!) и определенный порядок увеличения и уменьшения количества pod (scale-up, scale-down). Это не так важно, когда развертывается одноэлементное приложение, но если вам нужно развернуть состояние с репликацией, то данные характеристики придутся очень кстати.

Чтобы запросить постоянный том для нашего сервиса Redis, мы воспользуемся `PersistentVolumeClaim`. Это своеобразный запрос ресурсов. Наш сервис объявляет, что ему нужно хранилище размером 50 Гбайт, а кластер Kubernetes определяет, как выделить подходящий постоянный том. Данный механизм нужен по двум причинам. Во-первых, он позволяет создать ресурс `StatefulSet`, который можно переносить между разными облаками и размещать локально, не заботясь о конкретных физических дисках. Во-вторых, несмотря на то, что том типа `PersistentVolume` можно подключить лишь к одному pod, запрос тома позволяет написать шаблон, доступный для реплицирования, но при этом каждому pod будет назначен отдельный постоянный том.

Ниже показан пример ресурса `StatefulSet` для Redis с постоянными томами:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis"
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
```

```
    spec:
      containers:
      - name: redis
        image: redis:5-alpine
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - name: data
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 10Gi
```

В результате будет развернут один экземпляр сервиса Redis. Но, допустим, вам нужно реплицировать кластер Redis, чтобы масштабировать запросы на чтение и повысить устойчивость к сбоям. Для этого, очевидно, следует довести количество реплик до трех, но в то же время сделать так, чтобы для выполнения записи новые реплики подключались к ведущему экземпляру Redis.

Когда мы добавляем в объект `StatefulSet` новый неуправляемый (headless) сервис, для него автоматически создается DNS-запись `redis-0.redis`; это IP-адрес первой реплики. Вы можете воспользоваться этим для написания сценария, пригодного для запуска во всех контейнерах:

```
#!/bin/sh

PASSWORD=$(cat /etc/redis-passwd/passwd)

if [[ "${HOSTNAME}" == "redis-0" ]]; then
  redis-server --requirepass ${PASSWORD}
else
  redis-server --slaveof redis-0.redis 6379 --masterauth ${PASSWORD}
  --requirepass ${PASSWORD}
fi
```

Этот сценарий можно оформить в виде `ConfigMap`:

```
kubectl create configmap redis-config --from-file=launch.sh=launch.sh
```

Затем объект `ConfigMap` нужно добавить в `StatefulSet` и использовать его как команду для управления контейнером. Добавим также пароль для аутентификации, который создали ранее.

Полное определение сервиса Redis с тремя репликами выглядит следующим образом:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis"
  replicas: 3
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:5-alpine
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - name: data
          mountPath: /data
        - name: script
          mountPath: /script/launch.sh
          subPath: launch.sh
        - name: passwd-volume
          mountPath: /etc/redis-passwd
        command:
        - sh
        - -c
        - /script/launch.sh
      volumes:
      - name: script
        configMap:
          name: redis-config
          defaultMode: 0777
      - name: passwd-volume
        secret:
          secretName: redis-passwd
  volumeClaimTemplates:
  - metadata:
      name: data
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 10Gi
```

Создание балансировщика нагрузки для TCP с использованием Service

Итак, мы развернули stateful-сервис Redis; теперь его нужно сделать доступным для нашего клиентского приложения. Для этого создадим два разных `Service` Kubernetes. Первый будет читать данные из Redis. Поскольку они реплицируются между всеми тремя участниками `StatefulSet`, для нас несущественно, к какому из них будут направляться наши запросы на чтение. Следовательно, для этой задачи подойдет простой `Service`:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis
  name: redis
  namespace: default
spec:
  ports:
  - port: 6379
    protocol: TCP
    targetPort: 6379
  selector:
    app: redis
  sessionAffinity: None
  type: ClusterIP
```

Выполнение записи потребует обращения к ведущей реплике Redis (под номером 0). Создайте для этого *неуправляемый* (headless) `Service`. У него нет IP-адреса внутри кластера; вместо этого он задает отдельную DNS-запись для каждого pod в `StatefulSet`. То есть мы можем обратиться к нашей ведущей реплике по доменному имени `redis-0.redis`:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis-write
  name: redis-write
spec:
  clusterIP: None
  ports:
  - port: 6379
  selector:
    app: redis
```

Таким образом, если нам нужно подключиться к Redis для сохранения каких-либо данных или выполнения транзакции с чтением/записью, то мы

можем собрать отдельный клиент, который будет подключаться к серверу `redis-0.redis-write`.

Направление трафика к серверу статических файлов с помощью Ingress

Заключительный компонент нашего приложения — *сервер статических файлов*, который отвечает за раздачу HTML-, CSS-, JavaScript-файлов и изображений. Отделение сервера статических файлов от нашего клиентского приложения, предоставляющего API, делает нашу работу более эффективной и целенаправленной. Для раздачи файлов можно воспользоваться готовым высокопроизводительным файловым сервером наподобие NGINX; при этом команда разработчиков может сосредоточиться на реализации нашего API.

К счастью, ресурс `Ingress` позволяет очень легко организовать такую архитектуру в стиле мини/микросервисов. Как и в случае с клиентским приложением, мы можем описать реплицируемый сервер NGINX с помощью ресурса `Deployment`. Соберем статические образы в контейнер NGINX и развернем их в каждой реплике. Ресурс `Deployment` будет выглядеть следующим образом:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: fileserver
  name: fileserver
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: fileserver
  template:
    metadata:
      labels:
        app: fileserver
    spec:
      containers:
      - image: my-repo/static-files:v1-abcde
        imagePullPolicy: Always
        name: fileserver
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        resources:
          requests:
            cpu: "1.0"
```

```
            memory: "1G"
          limits:
            cpu: "1.0"
            memory: "1G"
      dnsPolicy: ClusterFirst
      restartPolicy: Always
```

Теперь, запустив реплицируемый статический веб-сервер, вы можете аналогичным образом создать ресурс `Service`, который будет играть роль балансировщика нагрузки:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: fileserver
  name: fileserver
  namespace: default
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: fileserver
  sessionAffinity: None
  type: ClusterIP
```

Итак, у вас есть `Service` для сервера статических файлов. Добавим в ресурс `Ingress` новый путь. Необходимо отметить, что путь `/` должен идти *после* `/api`, иначе запросы API станут направляться серверу статических файлов. Обновленный ресурс `Ingress` будет выглядеть следующим образом:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: frontend-ingress
spec:
  rules:
  - http:
    paths:
    - path: /api
      backend:
        serviceName: frontend
        servicePort: 8080
    # Примечание: этот раздел должен идти после /api,
    # иначе он будет перехватывать запросы.
    - path: /
      backend:
        serviceName: fileserver
        servicePort: 80
```

Параметризация приложения с помощью Helm

Все, что мы обсуждали до сих пор, было направлено на развертывание одного экземпляра нашего сервиса в одном кластере. Но в реальности сервисы почти всегда приходится развертывать в нескольких разных средах (даже при условии, что они находятся в общем кластере). Вы можете быть разработчиком-одиночкой, который занимается всего одним приложением, но если хотите, чтобы внесение изменений не мешало пользователям работать, то вам понадобится как минимум две версии: отладочная и промышленная. И прибавив к этому интеграционное тестирование и CI/CD, мы получим следующее: даже при работе с одним сервисом и наличии лишь пары разработчиков нужно выполнять развертывание по меньшей мере в трех разных средах, и это далеко не предел, если приложение должно справляться со сбоями на уровне вычислительного центра.

Начальная стратегия для борьбы со сбоями у многих команд заключается в тривиальном копировании файлов из одного кластера в другой. Вместо одного каталога frontend/ они используют два: frontend-production/ и frontend-development/. Такой способ опасен, поскольку разработчикам приходится следить за тем, чтобы файлы оставались синхронизированными. Этого можно было бы легко добиться, если бы эти каталоги должны были быть идентичными. Но некоторые расхождения между отладочной и промышленной версиями нормальны, так как вы будете разрабатывать новые возможности; крайне важно, чтобы эти расхождения были намеренными и простыми в управлении.

Еще один подход состоит в использовании веток и системы контроля версий; центральный репозиторий разделяется на промышленную и отладочную ветки, разница между которыми видна невооруженным глазом. Это может быть хорошим вариантом для некоторых команд, но если вы хотите развертывать ПО сразу в нескольких средах (например, система CI/CD может выполнять развертывание в разных регионах облака), то переключение между ветками будет проблематичным.

В связи с этим большинство людей в итоге выбирают *систему шаблонов*. Идея в том, что централизованный каркас конфигурации приложения образуют шаблоны, которые *подставляются* для той или иной среды на основе параметров. Таким образом, вы можете иметь одну общую конфигурацию, при необходимости легко подгоняемую под определенные условия. Для Kubernetes есть множество разных систем шаблонов, но наиболее популярна, безусловно, Helm (helm.sh).

В Helm приложения распространяются в виде так называемых *чартов* с файлами внутри.

В основе чарта лежит файл `chart.yaml`, в котором определяются его метаданные:

```
apiVersion: v1
appVersion: "1.0"
description: A Helm chart for our frontend journal server.
name: frontend
version: 0.1.0
```

Этот файл размещается в корневом каталоге чарта (например, в frontend/). Там же находится каталог templates, внутри которого хранятся шаблоны. Шаблон, в сущности, представляет собой YAML-файл, похожий на приводимые в предыдущих примерах; разница лишь в том, что отдельные его значения заменены ссылками на параметры. Скажем, представьте, будто хотите параметризировать количество реплик в своем клиентском приложении. Вот что содержал наш исходный объект `Deployment`:

```
...
spec:
  replicas: 2
...
```

В файле шаблона (`frontend-deployment.tmpl`) данный раздел выглядит следующим образом:

```
...
spec:
  replicas: {{ .replicaCount }}
...
```

Это значит, что при развертывании чарта для поля `replicas` будет подставлен подходящий параметр. Сами параметры определены в файле `values.yaml`, предназначенном для конкретной среды, в котором развертывается приложение. Для этого простого чарта файл `values.yaml` выглядел бы так:

```
replicaCount: 2
```

Теперь, чтобы собрать все указанное вместе, вы можете развернуть данный чарт с помощью утилиты `helm`, как показано ниже:

```
helm install path/to/chart --values path/to/environment/values.yaml
```

Эта команда параметризирует ваше приложение и развернет его в Kubernetes. Со временем параметризация будет расширяться, охватывая все разнообразие сред выполнения вашего приложения.

Рекомендации по развертыванию сервисов

Kubernetes — эффективная система, которая может показаться сложной. Однако процесс развертывания обычного приложения легко упростить, если следовать общепринятым рекомендациям.

- Большинство сервисов нужно развертывать в виде ресурса `Deployment`. Объекты `Deployment` создают идентичные реплики для масштабирования и обеспечения избыточности.
- Для доступа к объектам `Deployment` можно использовать объект `Service`, который, в сущности, является балансировщиком нагрузки. `Service` может быть доступен как изнутри (по умолчанию), так и снаружи. Если вы хотите, чтобы к вашему HTTP-приложению можно было обращаться, то используйте контроллер `Ingress` для добавления таких возможностей, как маршрутизация запросов и SSL.
- Рано или поздно ваше приложение нужно будет параметризировать, чтобы сделать его конфигурацию более пригодной к использованию в разных средах. Для этого лучше всего подходят диспетчеры пакетов, такие как Helm (`helm.sh`).

Резюме

Несмотря на свою простоту, приложение, созданное нами в этой главе, охватывает практически все концепции, которые могут понадобиться вам в более крупных и сложных проектах. Понимание того, как сочетаются эти фундаментальные компоненты, и умение их использовать — залог успешного применения Kubernetes.

Использование системы контроля версий, аудита изменений кода и непрерывной доставки ваших сервисов позволит любым проектам, которые вы создаете, иметь прочный фундамент. Эта основополагающая информация пригодится вам при изучении более сложных тем, представленных в других главах.

ГЛАВА 2

Процесс разработки

Платформа Kubernetes была создана для надежной эксплуатации программного обеспечения. Она упрощает развертывание и администрирование приложений за счет API, ориентированного на управление программами, свойств самовосстановления и таких ценных инструментов, как объекты `Deployment`, которые позволяют развертывать ПО с нулевым временем простоя. И хотя все эти средства полезны, они не слишком помогают в разработке приложений для Kubernetes. Многие кластеры предназначены для выполнения промышленных приложений, поэтому редко участвуют в процессе разработки. Тем не менее крайне важно, чтобы разработчики имели возможность писать код с расчетом на Kubernetes, а это обычно подразумевает выделение кластера или как минимум некой его части под разработку. Это один из ключевых моментов в создании успешных приложений для данной платформы. Очевидно, что сам по себе кластер, для которого не собрано никакого кода, довольно бесполезен.

Цели

Прежде чем переходить к описанию лучших подходов к построению кластеров для разработки, следует обозначить цели, которые мы ставим перед собой. Очевидно, что основная цель — дать возможность разработчикам быстро и легко собирать приложения для Kubernetes, но что это означает на самом деле и какие практические требования к кластеру для разработки влечет за собой?

Определим этапы взаимодействия разработчика с кластером.

Все начинается с *подготовки*. Это когда к команде присоединяется новый разработчик. На данном этапе ему выдают учетную запись для доступа к кластеру и показывают, как производится развертывание. Цель этапа — за минимальные сроки подготовить разработчика к выполнению его обязанностей. Для этого следует определить ключевые показатели эффективно-

сти (key performance indicator, KPI), на которые нужно ориентироваться. Например, если пользователю удастся развернуть с нуля приложение из текущей ветки в течение получаса, то это можно считать успехом. Проверяйте эффективность данного этапа каждый раз, когда к команде присоединяется новый человек.

Второй этап — *разработка*. Это ежедневные обязанности разработчика. Цель этапа — быстрое развитие и отладка проекта. Разработчикам нужно быстро и по многу раз в день доставлять код в кластер. Они также должны иметь возможность легко тестировать свой код и отлаживать его в случае некорректной работы. Показатели эффективности данного этапа не так-то просто измерить, но вы можете их оценить, посчитав время, уходящее на развертывание в кластере PR (pull request — запрос на включение внесенных изменений) или изменения. Вместо этого (или в дополнение) уместно провести исследование субъективной производительности пользователей, которую также можно измерить в контексте общей производительности вашей команды.

Третий этап — *тестирование*. Он переплетается с разработкой и используется для проверки кода перед его загрузкой и слиянием. Цель этапа состоит из двух частей. Во-первых, разработчик должен иметь возможность выполнить все тесты для своей среды, прежде чем отправлять PR. Во-вторых, перед слиянием кода в репозитории все тесты должны запускаться автоматически. Вдобавок следует установить KPI для продолжительности выполнения тестов. Вполне естественно, что по мере того, как ваш проект усложняется, увеличивается количество его тестов и, следовательно, время их работы. В этом случае имеет смысл выделить некую часть тестов, которые разработчик может использовать для начальной проверки перед отправкой PR. Кроме того, следует определить очень жесткий KPI для *стабильности тестов*. Нестабильным называют тест, который изредка (или сравнительно часто) дает сбой. В любом относительно активном проекте нестабильность, выражающаяся в одном сбое на каждые тысячу запусков, приводит к разногласиям между разработчиками. Вы должны убедиться в том, что среда выполнения вашего кластера не является причиной нестабильности тестов. Тесты могут становиться нестабильными как из-за проблем в коде, так и вследствие некорректного функционирования среды разработки (например, при нехватке ресурсов или слишком активном поведении других тестов на том же оборудовании). Вам следует позаботиться о том, чтобы в среде разработки не было подобных проблем; для этого нужно отслеживать нестабильные тесты и быстро принимать соответствующие меры.

Построение кластера для разработки

Когда люди начинают задумываться о разработке для Kubernetes, одно из первых решений, которые им приходится принимать, заключается в выборе между созданием одного большого кластера для разработки и выделением отдельных кластеров для каждого разработчика. Стоит отметить, что этот выбор встает только в средах, позволяющих легко и динамически создавать новые кластеры (например, в публичных облаках). В физических окружениях один большой кластер может быть единственным вариантом.

Если у вас все же есть выбор, то вы должны подумать о преимуществах и недостатках каждого из двух решений. Можно выделить по одному кластеру для каждого разработчика, но существенным минусом этого подхода будут его высокая цена и низкая эффективность; к тому же вам придется управлять большим количеством разных сред для разработки. Дополнительные денежные расходы обусловлены тем, что существенная часть ресурсов каждого кластера будет простаивать. Кроме того, большое количество разных кластеров затруднит отслеживание и освобождение ненужных ресурсов. Преимущество этого подхода состоит в его простоте: каждый разработчик может самостоятельно заниматься обслуживанием своего кластера, а благодаря изоляции разные члены команды не будут мешать друг другу.

С другой стороны, единый кластер для разработки будет намного эффективней; цена обслуживания того же количества разработчиков, вероятно, будет как минимум втрое ниже. Кроме того, это значительно упрощает установку общих кластерных сервисов, таких как мониторинг и ведение журнала, благодаря чему намного легче сделать кластер удобным для разработки. Недостатки разделяемого кластера — процесс управления пользователями и недостаточная изоляция между разработчиками. В настоящее время процедуру добавления в Kubernetes новых пользователей и пространств имен нельзя назвать слишком простой. Механизмы для взаимодействия с ресурсами и управления доступом на основе ролей (Role-Based Access Control, RBAC), встроенные в Kubernetes, могут уменьшить риск возникновения конфликтов между двумя разработчиками. Однако всегда существует вероятность того, что пользователь займет слишком много ресурсов и заблокирует работу других приложений, в результате чего *нарушится* функционирование кластера. Кроме того, вам нужно следить за тем, чтобы разработчики не забывали освобождать выделенные им ресурсы, хотя по сравнению с подходом, в котором они создают собственные кластеры, это не так уж сложно.

Оба описанных варианта приемлемы, но в целом мы рекомендуем использовать единый большой кластер. Потенциальные конфликты между разными разработчиками устранимы, а экономичность и возможность легко добавлять в кластер новые функции в масштабах всей организации перевешивают вероятные риски. Но при этом необходимо позаботиться о подготовке условий для новых сотрудников, об управлении ресурсами и удалении ненужных сред. Мы считаем, что сначала следует попробовать один большой кластер. По мере роста вашей организации (или если она уже большая) можно будет подумать о выделении отдельных кластеров для каждой команды или группы (размером 10–20 человек), чтобы не сосредотачивать сотни пользователей в одной системе. Это может помочь как с планированием расходов, так и с управлением.

Подготовка разделяемого кластера для нескольких разработчиков

Крупный кластер создается для того, чтобы в нем могли работать сразу несколько пользователей, не мешая друг другу. Очевидным средством разделения разработчиков в Kubernetes служат пространства имен — на их основе можно создавать отдельные среды для развертывания, чтобы клиентские сервисы разных пользователей не конфликтовали между собой. Пространства имен также играют роль областей доступа для RBAC, благодаря которым разработчик не может случайно удалить чужие контейнеры. Таким образом, в разделяемом кластере пространства имен логично использовать в качестве рабочих областей. Процедуры добавления новых пользователей и создания/защиты пространств имен описываются в следующих подразделах.

Добавление новых пользователей

Прежде чем назначить пользователю пространство имен, вы должны дать доступ к самому кластеру Kubernetes. Вы можете создать для пользователя новый сертификат и предоставить ему файл `kubeconfig`, который позволит входить в систему; вы также можете настроить свой кластер в целях применения внешней системы идентификации (такой как Microsoft Azure Active Directory или AWS Identity and Access Management, IAM).

В целом применение внешней системы идентификации — предпочтительный вариант, поскольку она не требует поддержания двух разных источников

идентичности. Однако в случаях, когда она неприменима, приходится использовать сертификаты. К счастью, для создания и администрирования таких сертификатов в Kubernetes есть специальный API. Ниже описана процедура добавления нового пользователя в существующий кластер.

Первым делом нужно выполнить запрос для создания нового сертификата. Ниже представлена простая программа на Go, которая делает это:

```
package main

import (
        "crypto/rand"
        "crypto/rsa"
        "crypto/x509"
        "crypto/x509/pkix"
        "encoding/asn1"
        "encoding/pem"
        "os"
)

func main() {
        name := os.Args[1]
        user := os.Args[2]

        key, err := rsa.GenerateKey(rand.Reader, 1024)
        if err != nil {
                panic(err)
        }
        keyDer := x509.MarshalPKCS1PrivateKey(key)
        keyBlock := pem.Block{
                Type:  "RSA PRIVATE KEY",
                Bytes: keyDer,
        }
        keyFile, err := os.Create(name + "-key.pem")
        if err != nil {
                panic(err)
        }
        pem.Encode(keyFile, &keyBlock)
        keyFile.Close()
        commonName := user
        // подставьте сюда свои данные
        emailAddress := "someone@myco.com"

        org := "My Co, Inc."
        orgUnit := "Widget Farmers"
        city := "Seattle"
        state := "WA"
        country := "US"
```

```
        subject := pkix.Name{
                CommonName:         commonName,
                Country:            []string{country},
                Locality:           []string{city},
                Organization:       []string{org},
                OrganizationalUnit: []string{orgUnit},
                Province:           []string{state},
        }

        asn1, err := asn1.Marshal(subject.ToRDNSequence())
        if err != nil {
                panic(err)
        }
        csr := x509.CertificateRequest{
                RawSubject:         asn1,
                EmailAddresses:     []string{emailAddress},
                SignatureAlgorithm: x509.SHA256WithRSA,
        }

        bytes, err := x509.CreateCertificateRequest(rand.Reader, &csr, key)
        if err != nil {
                panic(err)
        }
        csrFile, err := os.Create(name + ".csr")
        if err != nil {
                panic(err)
        }

        pem.Encode(csrFile, &pem.Block{Type: "CERTIFICATE REQUEST", Bytes: bytes})
        csrFile.Close()
}
```

Вы можете запустить ее следующим образом:

```
go run csr-gen.go client <user-name>;
```

В результате будут созданы файлы с названиями `client-key.pem` и `client.csr`. Затем можно запустить следующий сценарий, чтобы создать и загрузить новый сертификат:

```
#!/bin/sh

csr_name="my-client-csr"

name="${1:-my-user}"

csr="${2}"

cat <<EOF | kubectl create -f -
apiVersion: certificates.k8s.io/v1beta1
```

```
kind: CertificateSigningRequest
metadata:
  name: ${csr_name}
spec:
  groups:
  - system:authenticated
  request: $(cat ${csr} | base64 | tr -d '\n')
  usages:
  - digital signature
  - key encipherment
  - client auth
EOF

echo
echo "Approving signing request."
kubectl certificate approve ${csr_name}

echo
echo "Downloading certificate."
kubectl get csr ${csr_name} -o jsonpath='{.status.certificate}' \
        | base64 --decode > $(basename ${csr} .csr).crt

echo
echo "Cleaning up"
kubectl delete csr ${csr_name}

echo
echo "Add the following to the 'users' list in your kubeconfig file:"
echo "- name: ${name}"
echo "  user:"
echo "    client-certificate: ${PWD}/$(basename ${csr} .csr).crt"
echo "    client-key: ${PWD}/$(basename ${csr} .csr)-key.pem"
echo
echo "Next you may want to add a role-binding for this user."
```

Сценарий выводит итоговую информацию, которую можно добавить в файл `kubeconfig` для подключения новой учетной записи. Конечно, созданный пользователь не имеет никаких прав доступа, поэтому вы должны применить к нему Kubernetes RBAC, чтобы он мог работать в заданном пространстве имен.

Создание и защита пространства имен

Чтобы выделить пространство имен, его нужно сначала создать. Для этого можно воспользоваться командой `kubectl create namespace my-namespace`.

Однако не все так просто. При создании пространства имен к нему необходимо прикрепить разные метаданные — например, контактную информа-

цию команды, которая развертывает в нем свои компоненты. В целом это имеет вид аннотаций; вы можете либо сгенерировать YAML-файл на основе какой-либо системы шаблонов наподобие Jinja (oreil.ly/vvtTF), либо создать пространство имен и самостоятельно добавить к нему аннотации. Для этого можно использовать простой сценарий:

```
ns='my-namespace'
kubectl create namespace ${ns}
kubectl annotate namespace ${ns} annotation_key=annotation_value
```

Создав пространство имен, вы должны его обезопасить, выдавая доступ к нему только определенным пользователям. Чтобы это сделать, пользователю можно назначить роль в контексте данного пространства. Сначала внутри `my-namespace` нужно создать объект `RoleBinding` примерно такого вида:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: example
  namespace: my-namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: edit
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: myuser
```

Чтобы его создать, выполните команду `kubectl create -f role-binding.yaml`. Обратите внимание: эту привязку можно использовать многократно; достаточно лишь указать в поле `namespace` подходящее пространство имен. Отсутствие у пользователя других привязок гарантирует, что данное пространство имен — единственная часть кластера, к которой он имеет доступ. Кроме того, пользователю также имеет смысл выдать доступ на чтение всего кластера; благодаря этому разработчики смогут следить за действиями своих коллег, если те влияют на их работу. Но будьте при этом осторожны: такой доступ на чтение, помимо прочего, позволяет читать секретные ресурсы кластера. В принципе, если тот используется только для разработки, то в этом нет ничего плохого, поскольку все пользователи трудятся в одной организации, а секретные данные все равно годятся лишь для разработки. Но если вас это беспокоит, то вы можете создать более гибкую роль, которая исключает чтение конфиденциальной информации.

Ограничить общее количество ресурсов, доступных конкретному пространству имен, можно с помощью объекта `ResourceQuota`. Например, следующая квота выделяет для всех pod в пространстве имен десять ядер и 100 Гбайт оперативной памяти (сразу для `Request` и `Limit`):

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: limit-compute
  namespace: my-namespace
spec:
  hard:
    requests.cpu: "10"
    requests.memory: 100Gi
    limits.cpu: "10"
    limits.memory: 100Gi
```

Управление пространствами имен

Вы уже увидели, как подключить нового пользователя и создать для него пространство имен, в котором он может работать. Но вопрос о способе назначения разработчика этому пространству по-прежнему остается открытым. Как часто бывает, у данной проблемы нет одного идеального решения, зато есть два разных подхода. Первый состоит в том, чтобы на подготовительном этапе выдавать каждому новому пользователю отдельное пространство имен. Это удобно, поскольку после подключения пользователь получает собственную рабочую среду, в которой может разрабатывать и администрировать свои приложения. Но если сделать данную среду слишком постоянной, то разработчик утратит мотивацию приводить свое пространство имен в порядок по окончании работы, что осложнит освобождение и учет отдельных ресурсов. В качестве альтернативного подхода пространствам имен можно назначать определенное время жизни (time to live, TTL). Благодаря этому разработчики будут считать ресурсы кластера временными и непостоянными; к тому же вам будет легче автоматизировать удаление целых пространств имен по истечении их TTL.

В этой модели перед началом нового проекта разработчик использует инструмент для выделения соответствующего пространства имен. Каждое пространство имеет метаданные, предназначенные для администрирования и учета ресурсов. Помимо TTL, эти метаданные могут включать в себя ID разработчика, которому назначено пространство, ресурсы, которые следует выделить (например, процессор и память), сведения о команде и описание

пространства. Благодаря этому вы можете отслеживать потребление ресурсов и удалить пространство имен в нужный момент.

Разработка инструментария для динамического выделения пространств имен может показаться сложной задачей, но в простых случаях с этим не должно быть никаких проблем. Например, можно использовать элементарный сценарий, который создает пространство имен и запрашивает подходящие метаданные, которые к нему нужно прикрепить.

Если вам нужна более тесная интеграция с Kubernetes, то можете применить пользовательские определения ресурсов (custom resource definitions, CRD), чтобы разработчики могли динамически создавать и выделять новые пространства имен с помощью утилиты `kubectl`. При наличии времени и желания этот подход, несомненно, себя оправдает, поскольку делает управление пространствами имен декларативным и позволяет использовать Kubernetes RBAC.

Ваш инструментарий должен предусматривать не только выделение новых, но и удаление существующих пространств имен с истекшим временем жизни. Опять же для этого достаточно простого сценария, который будет анализировать ваш кластер и удалять из него просроченные пространства.

Данный сценарий можно оформить в виде контейнера и запускать его с некой периодичностью (например, каждый час), используя `ScheduledJob`. В совокупности эти средства позволят разработчикам легко выделять ресурсы для отдельных проектов по мере необходимости; но в то же время ресурсы будут время от времени освобождаться, чтобы кластер всегда оставался эффективным и готовым к разработке новых проектов.

Сервисы уровня кластера

Помимо инструментария для выделения и администрирования пространств имен, вы можете использовать в своей разработке сервисы уровня кластера. Один такой сервис позволяет собирать журнальные записи в центральной системе LaaS (Logging as a Service — журналирование как услуга). Чтобы разобраться в поведении своего приложения, разработчику зачастую проще всего записать некую информацию в поток STDOUT. Но журнал, в который она попадет (и который можно просматривать с помощью команды `kubectl logs`), имеет ограниченную длину, и в нем сложно найти что-либо. Если же его содержимое автоматически передавать системе LaaS (в качестве которой может выступать облачный сервис или кластер Elasticsearch), то разработчикам

будет удобно собирать журнальные записи из разных контейнеров их сервиса и искать в них нужную им информацию.

Рабочие процессы разработчика

Итак, у нас есть общий кластер, к которому мы можем подключать новых разработчиков. Теперь нам нужно неким образом организовать разработку самих приложений. Как вы помните, один из ключевых показателей эффективности, который мы замеряем, — это то, насколько быстро новый пользователь может развернуть приложение в кластере. Очевидно, что мы можем быстро аутентифицировать разработчика в кластере и выделить ему пространство имен, используя описанные выше сценарии, но как насчет самой разработки? К сожалению, хоть этот процесс и можно облегчить с помощью нескольких методик, начальный процесс развертывания и запуска приложения опирается скорее на общепринятые правила, а не на автоматизацию. В следующих подразделах будет описан один (и далеко не единственный) подход к выполнению данной задачи. Вы можете применять его как есть или разработать на его основе собственное решение.

Начальная подготовка

Одна из главных трудностей при развертывании приложения — установка всех его зависимостей. Во многих случаях, особенно в современных микросервисных архитектурах, прежде чем начать работу над каким-либо одним компонентом, необходимо сначала развернуть его многочисленные зависимости, такие как базы данных или микросервисы. И хотя развертывание самого приложения — относительно простой процесс, в его сборке участвуют разные компоненты, подбирать которые зачастую приходится методом проб и ошибок, руководствуясь неполными или устаревшими инструкциями.

Справиться с данной проблемой можно с помощью введения правил по описанию и установке зависимостей. Это может быть эквивалентом такой команды, как `npm install`, устанавливающей все необходимые пакеты JavaScript. Когда-нибудь появится аналог `npm`, который будет предоставлять подобные возможности для приложений, основанных на Kubernetes, но в настоящий момент лучше всего выработать общие правила внутри своей организации.

Например, вы можете условиться о том, что в корневом каталоге каждого проекта должен находиться сценарий `setup.sh`, который будет отвечать за

корректное создание всех зависимостей приложения внутри определенного пространства имен. Он может выглядеть следующим образом:

```
kubectl create my-service/database-stateful-set-yaml
kubectl create my-service/middle-tier.yaml
kubectl create my-service/configs.yaml
```

Впоследствии этот сценарий можно интегрировать в `npm`, добавив следующий код в `package.json`:

```
{
    ...
    "scripts": {
        "setup": "./setup.sh",
        ...
    }
}
```

В такой среде для установки зависимостей разработчику достаточно выполнить команду `npm run setup`. Конечно, этот пример касается конкретно Node.js/npm. В других языках программирования нужно будет использовать другой инструментарий. Например, в Java установку зависимостей можно интегрировать в файл Maven `pom.xml`.

Активная разработка

Вслед за развертыванием зависимостей в рабочем пространстве следует сделать так, чтобы разработчик мог быстро выкатывать свои обновления. Необходимое условие для этого — возможность собирать и загружать образы контейнеров. Будем считать, что эту часть рабочего процесса вы уже выполнили (если есть сложности, то обратитесь к дополнительным источникам — этой теме посвящено множество онлайн-ресурсов и книг).

После сборки и загрузки образ контейнера необходимо развернуть в кластере. В отличие от традиционного развертывания, на этапе разработки нас не заботит поддержка непрерывной доступности приложения. Как следствие, чтобы выкатить код, проще всего удалить объект `Deployment`, относящийся к предыдущему развертыванию, и создать вместо него новый, который ссылается на собранный вами образ. Объекты `Deployment` можно также обновлять, но это инициирует логику развертывания обновлений в соответствующем ресурсе. В принципе, процесс развертывания кода можно сделать быстрым, но это приведет к расхождениям между отладочной и промышленной средами, что может иметь опасные последствия и плохо сказаться на стабильности.

Представьте, к примеру: вы случайно загрузили отладочную конфигурацию объекта `Deployment` в промышленную ветку репозитория; в результате ваши новые версии, не прошедшие надлежащего тестирования, неожиданно начнут работать сразу в промышленной среде, минуя остальные этапы развертывания. Учитывая данный риск и наличие альтернативы, объекты `Deployment` лучше удалять и создавать заново.

Для выполнения этого развертывания, как и для установки зависимостей, рекомендуется создать сценарий. В качестве примера можно привести такой файл под названием `deploy.sh`:

```
kubectl delete -f ./my-service/deployment.yaml
perl -pi -e 's/${old_version}/${new_version}/' ./my-service/deployment.yaml
kubectl create -f ./my-service/deployment.yaml
```

Как и прежде, этот код можно интегрировать с уже имеющимся инструментарием языка программирования, чтобы для развертывания своего нового кода в кластере разработчику было достаточно выполнить одну команду — например, `run deploy`.

Тестирование и отладка

После того как пользователь успешно развернул отладочную версию своего приложения, ему нужно проверить его и в случае каких-либо проблем провести отладку. В Kubernetes это тоже может вызвать затруднения, поскольку не всегда понятно, как именно взаимодействовать с кластером. Здесь можно использовать универсальную утилиту командной строки `kubectl`, которая поддерживает такие действия, как `kubectl logs`, `kubectl exec` и `kubectl port-forward` и т. д., но, чтобы научиться ею пользоваться и разобраться со всеми ее параметрами, нужен довольно серьезный опыт. Более того, поскольку она выполняется в терминале, для одновременного просмотра исходного кода и вывода запущенного приложения часто приходится открывать несколько окон.

Чтобы сделать процесс тестирования и отладки более простым и удобным, инструментарий Kubernetes все теснее интегрируется в среды разработки. В качестве примера можно привести открытое расширение Visual Studio (VS) Code for Kubernetes, которое можно бесплатно установить из магазина VS Code. Оно автоматически распознает все кластеры в вашем файле `kubeconfig` и предоставляет навигационную панель с древовидным представлением их содержимого.

Помимо возможности просматривать состояние кластеров, интеграция позволяет разработчику использовать инструменты, поддерживаемые утилитой `kubectl`, с помощью интуитивного и удобного интерфейса. Если щелкнуть правой кнопкой мыши на pod Kubernetes в древовидном представлении, к нему сразу же можно подключиться с локального компьютера путем перенаправления портов. Точно так же можно получить доступ к журнальным записям pod или даже открыть терминал в контексте его активного контейнера.

Интеграция указанных команд со стандартными элементами пользовательского интерфейса (такими как контекстное меню), а также интеграция этих элементов с кодом самого приложения позволяет разработчикам, имеющим минимальный опыт использования Kubernetes, быстро освоиться в кластере для разработки.

Конечно, это расширение для VS Code — далеко не единственный пример интеграции Kubernetes и сред разработки; вы можете воспользоваться и другими инструментами, предназначенными для разных редакторов кода (`vi`, `emacs` и т. д.).

Рекомендации по подготовке среды для разработки

Налаживание успешного рабочего процесса в Kubernetes — залог подуктивности и гармонии. Изложенные ниже рекомендации помогут организовать вашу среду так, чтобы разработчики могли быстро приступать к своим обязанностям.

- Разделите процесс на три этапа: подготовка, разработка, тестирование. Убедитесь в том, что все они поддерживаются созданной вами средой для разработки.
- При построении отладочной среды используйте как один большой кластер, так и множество мелких, выделяемых для каждого отдельного разработчика. У обоих подходов есть свои плюсы и минусы, но в целом более предпочтителен единый общий кластер.
- Каждый пользователь, которого вы добавляете в кластер, должен иметь свою учетную запись и доступ к собственному пространству имен. Задействуйте лимиты, чтобы ограничить ресурсы кластера, которые доступны пользователям.
- При управлении пространствами имен не забывайте об освобождении старых, неиспользуемых ресурсов. Разработчики, как правило, не слишком

заботятся об удалении вещей, которые им больше не нужны. Применяйте автоматизацию, чтобы делать это за них.

- Подумайте о сервисах уровня кластера, таких как журналирование и мониторинг, которые можно сделать общими для всех разработчиков. Кроме того, иногда имеет смысл задействовать шаблоны наподобие Helm-чартов, чтобы предоставить общекластерные зависимости всем пользователям сразу.

Резюме

На сегодняшний день создание кластеров Kubernetes, особенно в облаке, — относительно тривиальная задача, но разработчики смогут продуктивно использовать эти кластеры только при наличии куда менее очевидных и простых решений. Если вы хотите, чтобы разработчики могли успешно создавать приложения в Kubernetes, вам необходимо подумать о ключевых целях, на которые нужно ориентироваться на таких этапах, как подготовка, редактирование кода, тестирование и отладка приложений. Точно так же стоит позаботиться о базовых инструментах для подготовки рабочей среды пользователя, выделении пространств имен и подключении общекластерных сервисов, таких как агрегация журнальных записей. Воспринимайте кластер для разработки и репозитории с кодом как средства для стандартизации ваших рабочих процессов и применения рекомендованных методик. Это поможет вашим разработчикам успешно собирать свой код для развертывания в промышленных средах Kubernetes.

ГЛАВА 3

Мониторинг и ведение журнала в Kubernetes

В этой главе мы обсудим рекомендуемые подходы к мониторингу и журналированию в Kubernetes. Мы подробно рассмотрим разные методы мониторинга, важные метрики, которые стоит собирать, и процесс создания информационных панелей на основе этих собранных метрик. В конце будут представлены примеры реализации системы мониторинга в кластере Kubernetes.

Метрики и журнальные записи

Для начала нужно разобраться в различиях между сбором журнальных записей и метрик. Эти данные дополняют друг друга, но имеют разное назначение.

- *Метрики*. Последовательность числовых показателей, замеренных на каком-либо отрезке времени.
- *Журнальные записи*. Используются для разведочного анализа системы.

Плохая производительность приложения — пример ситуации, когда вам могут потребоваться как метрики, так и журнальные записи. Первым признаком проблемы может быть уведомление о высокой задержке выполнения приложений в pod, но, чтобы как следует разобраться в случившемся, одних метрик может быть недостаточно. Поэтому можно обратиться к журнальным записям и поискать сообщения об ошибках, которые выдает приложение.

Разновидности мониторинга

Мониторинг таких компонентов, как процессоры, оперативная память, накопители и т. д., традиционно осуществляется с точки зрения внешнего наблюдателя по принципу черного ящика (black-box monitoring). Тот же принцип можно применять и для мониторинга на инфраструктурном уровне, но это не позволяет получить достаточно информации о работе приложения. Например, чтобы проверить работоспособность кластера, можно попробовать развернуть

pod; если нам это удастся, то мы будем знать, что планировщик и механизм обнаружения сервисов внутри нашего кластера работают исправно, поэтому можно предположить, что и с другими компонентами все в порядке.

Мониторинг по принципу белого ящика (white-box monitoring) дает возможность оценивать работу приложения в контексте его состояния, позволяя узнать, сколько всего было выполнено HTTP-запросов, сколько из них вернуло ошибку с кодом 500, какова их задержка и т. д. Такой мониторинг дает возможность понять, *почему* система работает именно так, а не иначе. Например, если на диске закончилось свободное место, мы можем спросить, почему это произошло.

Методы мониторинга

Некоторые читатели могут сказать: «Неужели это так сложно? Мы всегда проводим мониторинг наших систем». Действительно, отдельные методы мониторинга, которые используются на сегодняшний день, подходят и для Kubernetes. Однако такие платформы, как Kubernetes, отличаются высокой динамичностью и коротким временем жизни своих ресурсов, поэтому для их мониторинга требуется другой образ мышления. Например, виртуальная машина (ВМ), как правило, работает круглосуточно и сохраняет свое состояние. Но в Kubernetes pod ведут себя очень динамично и существуют недолго, вследствие чего вам нужна система мониторинга, способная справиться с таким непостоянством.

Существует несколько разных подходов к мониторингу распределенных систем.

Метод *USE*, популяризованный Бренданом Греггом, сосредоточен на следующих аспектах:

- U — Utilization (использование);
- S — Saturation (загруженность);
- E — Errors (ошибки).

Этот метод в основном предназначен для мониторинга инфраструктуры и не очень хорошо подходит для отслеживания состояния отдельных приложений. Его можно сформулировать таким образом: «Проверяйте использование, степень загруженности и ошибки для каждого ресурса». Данный метод позволяет быстро определить, сколько ресурсов доступно вашей системе и как часто в ней возникают ошибки. Например, если вы хотите проверить

работоспособность сети в контексте узлов кластера, с помощью метода USE можно легко найти ее узкие места и ошибки, возникающие в сетевом стеке. Следует отметить, что это лишь один из механизмов, которые вы будете применять для мониторинга своих систем.

Еще один подход к мониторингу, получивший название *RED*, стал популярным благодаря Тому Уиллке. Он сосредоточен на следующих аспектах:

- R — Rate (частота);
- E — Errors (ошибки);
- D — Duration (продолжительность).

Этот метод основан на философии *четырех золотых сигналов*, которую практикуют в компании Google:

- латентность (сколько времени занимает обработка запроса);
- трафик (количество запросов к вашему приложению);
- ошибки (доля неудачных запросов);
- степень загруженности (насколько интенсивно используется ваш сервис).

Например, при мониторинге клиентского сервиса, запущенного в Kubernetes, этот подход позволяет узнать:

- сколько запросов обрабатывает клиентский сервис;
- сколько ошибок с кодом 500 получают пользователи сервиса;
- перегружен ли сервис запросами.

Как видите, этот метод больше сосредоточен на впечатлениях пользователей от работы с сервисом.

Методы USE и RED дополняют друг друга: первый предназначен для инфраструктурных компонентов, а второй нацелен на конечного пользователя и его взаимодействие с приложением.

Обзор метрик, доступных в Kubernetes

Итак, мы познакомились с разными видами и методами мониторинга. Теперь поговорим о том, за какими компонентами кластера Kubernetes необходимо следить. Кластер Kubernetes состоит из управляющих компонентов (control plane) и рабочих узлов. К управляющим компонентам относятся

API-сервер, etcd, планировщик и менеджер контроллеров. Рабочие узлы содержат kubelet, среду выполнения контейнера, kube-proxy, kube-dns и pod. Чтобы обеспечить работоспособность кластера и приложений, следует проводить мониторинг всех этих компонентов.

Кластер Kubernetes предоставляет эти метрики множеством разных способов, так что рассмотрим инструменты, которые позволяют их собирать.

cAdvisor

cAdvisor (Container Advisor) — проект с открытым исходным кодом, предназначенный для сбора ресурсов и метрик контейнеров, выполняющихся на узле. Он встроен в утилиту kubelet, которая работает на каждом узле кластера, и собирает информацию о памяти и процессоре, используя иерархию контрольных групп Linux (control groups, cgroups). Если вы еще незнакомы с понятием cgroup, то это функция ядра Linux, позволяющая изолировать ресурсы процессора, а также дисковый и сетевой ввод/вывод. cAdvisor собирает сведения о диске с помощью вызова `statfs`, встроенного в ядро Linux. Вам не следует беспокоиться об этих деталях реализации, но вы должны понимать, как предоставляются те или иные метрики и какого рода информацию можете собирать. cAdvisor применим в качестве источника всех метрик контейнера.

Сервер метрик

Kubernetes предоставляет сервер и API для сбора метрик, которые пришли на смену устаревшей системе Heapster. Данная система имела отдельные архитектурные недостатки в реализации приемника данных, поощрявшие добавление в его ядро кода, рассчитанного на определенных облачных провайдеров. Для решения этой проблемы в Kubernetes был реализованы два агрегированных API, Resource Metrics API и Custom Metrics API, которые позволяют прозрачно переходить с одной реализации на другую.

API-сервер метрик имеет две ключевые особенности.

Во-первых, каноническая реализация Resource Metrics API — сервер метрик, который собирает метрики таких ресурсов, как процессор и память. В качестве источника он использует API kubelet, сохраняя результаты в память. Собранные метрики применяются в планировщике и контроллерах для горизонтального и вертикального автомасштабирования (Horizontal Pod Autoscaler (HPA) и Vertical Pod Autoscaler (VPA)).

Во-вторых, Custom Metrics API позволяет собирать произвольные метрики. Таким образом, системы мониторинга могут предоставлять собственные адаптеры, расширяя тем самым стандартный набор метрик ресурсов. Например, один из первых подобных адаптеров появился в Prometheus, что позволило задействовать HPA на основе пользовательских метрик. Это делает возможным более гибкое масштабирование с учетом конкретной ситуации, поскольку теперь вы можете учитывать метрики, не встроенные в Kubernetes, — например, размер очереди.

Таким образом, наличие стандартного API для работы с метриками позволяет привязывать масштабирование не только к показателям процессора и памяти, но и к множеству других данных.

kube-state-metrics

kube-state-metrics — дополнение для Kubernetes, которое занимается мониторингом объектов, хранящихся в кластере. Если cAdvisor и сервер метрик используются для предоставления подробной информации о потреблении ресурсов, то kube-state-metrics помогает определить состояние объектов Kubernetes, развернутых в кластере.

Ниже представлены вопросы, на которые помогает ответить kube-state-metrics.

- Pod.
  - Сколько pod развернуто в кластере?
  - Сколько pod простаивает?
  - Достаточно ли ресурсов для обслуживания запросов pod?
- Развертывание.
  - Сколько активных pod находятся в желаемом состоянии?
  - Сколько реплик мне доступно?
  - Какие объекты `Deployment` были обновлены?
- Узлы.
  - В каком состоянии находятся мои рабочие узлы?
  - Какие ядра процессора можно выделить в моем кластере?
  - Есть ли в кластере узлы, на которых нельзя запустить pod?

- Задания.
  - Когда задание начало выполняться?
  - Когда оно завершилось?
  - Сколько заданий завершилось неудачно?

На момент написания этой книги kube-state-metrics умеет отслеживать 22 типа объектов, но данный показатель постоянно растет. Вы можете найти документацию в репозитории на GitHub (oreil.ly/bdTp2).

Какие метрики нужно отслеживать

Проще ответить «все», но если отслеживать слишком много метрик, то можно получить огромный набор информации, в котором будет сложно найти нужные сигналы. Подход к мониторингу в Kubernetes должен быть многоуровневым и охватывать следующие аспекты:

- физические или виртуальные узлы;
- компоненты кластера;
- дополнения к кластеру;
- приложения, взаимодействующие с конечными пользователями.

Это упрощает корректное определение сигналов в системе мониторинга. Вы можете подходить к решению проблем более целенаправленно. Например, при переходе ваших pod в состояние ожидания можете сначала проверить загруженность узлов, и если все в порядке, то перейти к компонентам кластера.

Ниже перечислены метрики, которые имеет смысл отслеживать в своей системе.

- Узлы:
  - использование процессора;
  - использование памяти;
  - использование сети;
  - использование диска.
- Компоненты кластера:
  - латентность etcd.

- Дополнения к кластеру:
  - контроллер автомасштабирования;
  - контроллер Ingress.
- Приложение:
  - использование и загруженность памяти в контейнере;
  - использование процессора в контейнере;
  - использование сети и частота возникновения сетевых ошибок в контейнере;
  - метрики, относящиеся к фреймворку приложения.

Средства мониторинга

Kubernetes поддерживает интеграцию со многими средствами мониторинга. Их количество растет каждый день, а совместимость их возможностей с Kubernetes постоянно улучшается. Ниже перечислено несколько популярных решений.

- *Prometheus* — открытый проект, который предоставляет систему мониторинга и набор инструментов для создания уведомлений. Был основан в 2012 году компанией SoundCloud, и с тех пор на него перешли многие организации. У проекта есть очень активные разработчики и сообщество пользователей. На сегодняшний день Prometheus — самостоятельный, открытый проект, который больше не имеет отношения ни к какой компании. Чтобы подчеркнуть это и прояснить управленческую структуру, в 2016 году Prometheus вслед за Kubernetes присоединился к Cloud Native Computing Foundation (CNCF).
- *InfluxDB* — база данных для хранения временных рядов, рассчитанная на высокие нагрузки в плане записи и выполнения запросов. Неотъемлемый компонент стека TICK (Telegraf, InfluxDB, Chronograf и Kapacitor), предназначена для использования в качестве резервного хранилища в любых ситуациях, в которых задействованы большие объемы данных с временными метками, включая мониторинг инфраструктуры, метрики приложений, показания датчиков в IoT и аналитику в режиме реального времени.

- *Datadog* — предоставляет услуги мониторинга для облачных приложений любого масштаба, включая мониторинг серверов, баз данных, инструментов и сервисов с использованием платформы для анализа данных на основе SaaS.
- *Sysdig Monitor* — коммерческое решение для контейнерных приложений, поддерживающее мониторинг Docker и Kubernetes. За счет тесной интеграции с Kubernetes также позволяет собирать, сравнивать и запрашивать показатели Prometheus.
- *Средства облачных провайдеров*.
  - *GCP Stackdriver*. Система Stackdriver Kubernetes Engine Monitoring предназначена для мониторинга кластеров в Google Kubernetes Engine (GKE). Кроме того, занимается ведением журнала для сервисов и предоставляет интерфейс с информационной панелью, адаптированной под нужды GKE. Система Stackdriver Monitoring позволяет следить за производительностью, продолжительностью работы серверов и общей работоспособностью облачных приложений. Она собирает показатели, события и метаданные из Google Cloud Platform (GCP), Amazon Web Services (AWS), локально размещенных механизмов для проверки времени работы и средств инструментирования приложений.
  - *Microsoft Azure Monitor для контейнеров* — одна из возможностей платформы Azure, предназначенная для мониторинга производительности контейнерных рабочих заданий, развернутых либо в Azure Container Instances, либо в кластерах Kubernetes, размещенных в Azure Kubernetes Service. Мониторинг контейнеров чрезвычайно важен, особенно в промышленном, крупномасштабном кластере с множеством приложений. Azure Monitor для контейнеров позволяет получить представление о производительности, собирая метрики памяти и процессоров из контроллеров, узлов и контейнеров, доступных в Kubernetes, используя Metrics API. При этом также собираются журнальные записи контейнеров. Все это делается автоматически после включения мониторинга в кластере Kubernetes с применением контейнерной версии агента Log Analytics для Linux.
  - *AWS Container Insights*. Если вы задействуете Amazon Elastic Container Service (ECS), Amazon Elastic Kubernetes Service или другие платформы Kubernetes в Amazon EC2, то можете собирать, агрегировать и подытоживать метрики и журнальные записи своих контейнерных

приложений и микросервисов с помощью CloudWatch Container Insights. В число доступных метрик входит использование таких ресурсов, как процессор, память, диск и сеть. Кроме того, эта система предоставляет диагностическую информацию (например, о сбоях при перезапуске контейнеров), чтобы вы могли быстро определять и решать возникающие проблемы.

При реализации средств мониторинга следует обратить внимание на такой важный аспект, как способ хранения метрик. Решения, предоставляющие базы данных с поддержкой временных рядов и пар «ключ — значение», позволяют хранить метрики с большим количеством атрибутов.

Переход на новые средства мониторинга требует денежных расходов и усилий по изучению новых технологий и их внедрению. Поэтому, прежде чем идти на такой шаг, подумайте, чем вас не устраивают уже имеющиеся инструменты. Многие системы мониторинга поддерживают интеграцию с Kubernetes, так что проверьте, какие из них вы уже используете и удовлетворяют ли они вашим требованиям.

Мониторинг в Kubernetes с использованием Prometheus

В этом разделе мы сосредоточимся на отслеживании метрик с помощью системы Prometheus, которая предоставляет хорошую интеграцию с метками, механизмом обнаружения сервисов и метаданными Kubernetes. Общие концепции, которые мы здесь реализуем, применимы и к другим решениям.

Prometheus — проект с открытым исходным кодом, который входит в состав CNCF. Изначально он разрабатывался компанией SoundCloud и во многом был вдохновлен внутренней системой мониторинга Google BorgMon. В нем реализована многоуровневая модель данных с парами «ключ — значение», которая по своему принципу работы очень напоминает механизм меток в Kubernetes. Prometheus предоставляет метрики в формате, понятном человеку. Например:

```
# HELP node_cpu_seconds_total Seconds the CPU is spent in each mode.
# TYPE node_cpu_seconds_total counter
node_cpu_seconds_total{cpu="0",mode="idle"} 5144.64
node_cpu_seconds_total{cpu="0",mode="iowait"} 117.98
```

Для сбора метрик сервер Prometheus применяет активную модель (pull), обращаясь за данными к соответствующей конечной точке. Этот процесс упрощается благодаря тому, что такие платформы, как Kubernetes, сами предоставляют доступ к своим метрикам в формате Prometheus. Многие другие проекты в экосистеме Kubernetes (NGINX, Traefik, Istio, LinkerD и т. д.) делают то же самое. Вдобавок Prometheus использует средства экспорта для преобразования метрик ваших сервисов в свой формат.

Как видно на рис. 3.1, эта система имеет очень простую архитектуру.

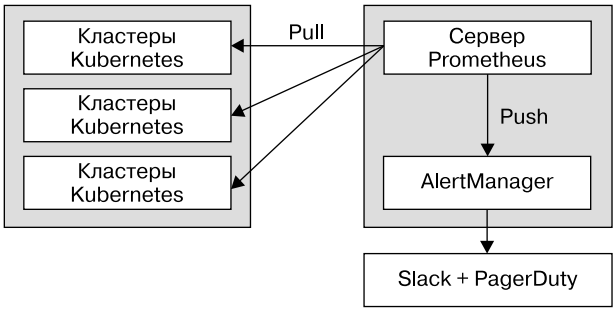


Рис. 3.1. Архитектура Prometheus

Prometheus можно установить как внутри кластера, так и за его пределами. Для мониторинга рекомендуется использовать отдельный «служебный кластер», чтобы на него не могли повлиять проблемы в промышленной среде. Обеспечить высокую доступность Prometheus и экспорт метрик во внешнюю систему хранения помогают такие инструменты, как Thanos (https://oreil.ly/7e6Wf).

Мы не станем глубоко погружаться в архитектуру Prometheus. Этой теме посвящены отдельные книги; можете начать с *Prometheus: Up & Running* (O'Reilly) (https://oreil.ly/NewNE).

Итак, подготовим Prometheus для работы в нашем кластере Kubernetes. Это можно сделать множеством разных способов, а процесс развертывания зависит от вашей конкретной реализации. В данной главе мы установим Prometheus Operator:

- *сервер Prometheus* — загружает и сохраняет метрики, собранные из разных систем;

- *Prometheus Operator* — интегрирует конфигурацию Prometheus в Kubernetes и управляет кластерами Prometheus и Alertmanager. Позволяет создавать, удалять и конфигурировать ресурсы Prometheus с помощью стандартных для Kubernetes определений ресурсов;
- *Node Exporter* — экспортирует метрики из узлов кластера Kubernetes;
- *kube-state-metrics* — собирает метрики, относящиеся к Kubernetes;
- *Alertmanager* — позволяет конфигурировать и направлять уведомления во внешние системы;
- *Grafana* — предоставляет возможность визуализации данных Prometheus с помощью информационных панелей.

```
helm install --name prom stable/prometheus-operator
```

После установки Operator в вашем кластере должны быть развернуты следующие pod:

```
$ kubectl get pods -n monitoring
NAME                                   READY   STATUS    RESTARTS   AGE
alertmanager-main-0                    2/2     Running   0          5h39m
alertmanager-main-1                    2/2     Running   0          5h39m
alertmanager-main-2                    2/2     Running   0          5h38m
grafana-5d8f767-ct2ws                  1/1     Running   0          5h39m
kube-state-metrics-7fb8b47448-k6j6g    4/4     Running   0          5h39m
node-exporter-5zk6k                    2/2     Running   0          5h39m
node-exporter-874ss                    2/2     Running   0          5h39m
node-exporter-9mtgd                    2/2     Running   0          5h39m
node-exporter-w6xwt                    2/2     Running   0          5h39m
prometheus-adapter-66fc7797fd-ddgk5    1/1     Running   0          5h39m
prometheus-k8s-0                       3/3     Running   0          5h39m
prometheus-k8s-1                       3/3     Running   0          5h39m
prometheus-operator-7cb68545c6-gm84j   1/1     Running   0          5h39m
```

Взглянем на Prometheus Server, чтобы понять, какие запросы следует использовать для извлечения метрик Kubernetes:

```
kubectl port-forward svc/prom-prometheus-operator-prometheus 9090
```

Эта команда создает тоннель к нашей локальной системе на порте 9090. Теперь мы можем открыть браузер и подключиться к серверу Prometheus по адресу http://127.0.0.1:9090.

На рис. 3.2 показана веб-страница, которую вы должны увидеть в случае успешного развертывания Prometheus в своем кластере.

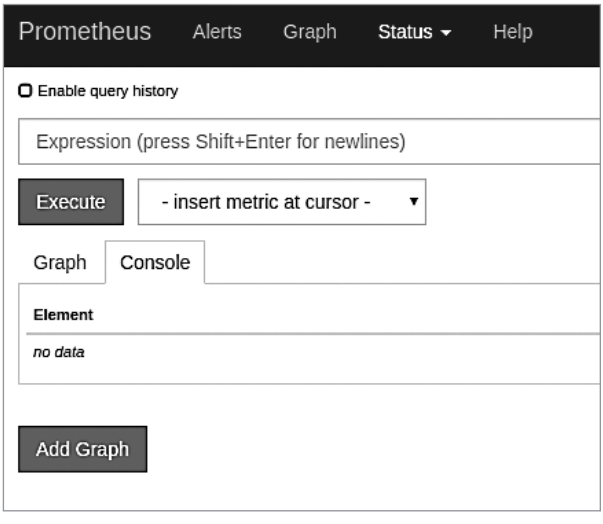


Рис. 3.2. Информационная панель Prometheus

Теперь исследуем какие-нибудь метрики Kubernetes, используя язык запросов Prometheus PromQL. Руководство по его основам доступно на странице oreil.ly/nGZYt.

Ранее в данной главе мы уже обсуждали применение метода USE, поэтому соберем некоторые метрики узлов, относящиеся к использованию и насыщению процессора.

Введите в поле Expression (Выражение) следующий запрос:

```
avg(rate(node_cpu_seconds_total[5m]))
```

В результате вы должны получить средний показатель использования процессоров во всем кластере.

Если нужен показатель использования процессора для каждого узла, то можно написать такой запрос:

```
avg(rate(node_cpu_seconds_total[5m])) by (node_name)
```

Результатом будет средний показатель использования процессоров для каждого узла в кластере.

Итак, вы получили некоторый опыт выполнения запросов в Prometheus. Теперь посмотрим, как создать информационную панель с визуализацией этих распространенных USE-метрик с помощью Grafana. Замечательной

особенностью Prometheus Operator является то, что вместе с этим инструментом устанавливаются информационные панели Grafana, уже готовые к использованию.

Чтобы вы могли обращаться к Grafana pod из своей локальной системы, вам необходимо создать тоннель с перенаправлением портов:

```
kubectl port-forward svc/prom-grafana 3000:3000
```

Теперь откройте в своем браузере http://localhost:3000 и войдите в систему, используя следующие учетные данные:

- имя пользователя: admin;
- пароль: admin.

На странице Grafana находится информационная панель под названием Kubernetes / USE Method / Cluster, которая облегчает просмотр таких ключевых метрик метода USE, как использование и загруженность кластера Kubernetes. Пример панели показан на рис. 3.3.

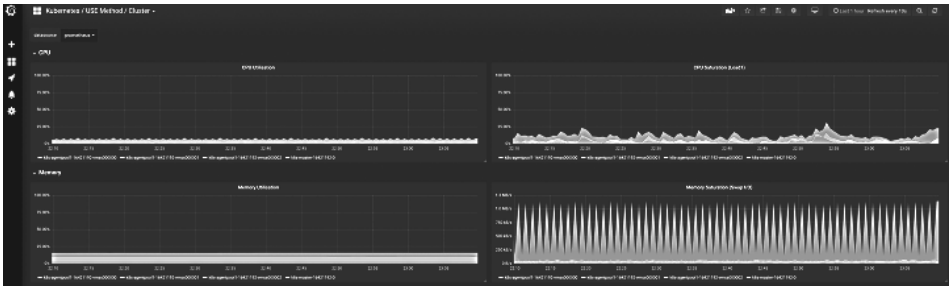


Рис. 3.3. Информационная панель Grafana

Уделите некоторое время исследованию разных информационных панелей и метрик, которые можно визуализировать в Grafana.

Не стоит создавать слишком много информационных панелей (так называемую стену графиков), поскольку во время диагностики проблем в них будет сложно разобраться. Можно было бы подумать, что заполненная информацией панель помогает в мониторинге, но большую часть времени она лишь создает путаницу. Проектируйте свои панели с прицелом на результат и минимизацию времени решения проблем.

Обзор журналирования

До сих пор мы уделяли много внимания метрикам и Kubernetes, но, помимо этого, для получения общей картины происходящего в вашей среде необходимо собирать и агрегировать журнальные записи как самого кластера, так и развернутых в нем приложений.

Вначале может показаться, что в журнал лучше записывать «все подряд», но в реальности это чревато двумя проблемами:

- ❑ слишком много лишней информации для быстрого поиска проблем;
- ❑ журнальные записи могут занимать много ресурсов и быть довольно затратными.

Нельзя однозначно сказать, что именно следует записывать в журнал. Со временем вы начинаете лучше разбираться в своей среде и понимать, от чего в системе журналирования можно избавиться. А чтобы справиться с постоянно растущим объемом сохраненных журнальных записей, вам нужно применять политику хранения и архивации. С точки зрения конечного пользователя, хороший компромисс — наличие записей за последние 30–45 дней. Это позволяет исследовать проблемы, проявляющиеся на протяжении длинных отрезков времени, но не требует слишком много ресурсов для хранения журналов. Если требования, предъявляемые к вашей системе, предусматривают долгосрочное хранение, то в целях экономии ресурсов журнальные записи лучше архивировать.

В кластере Kubernetes есть несколько компонентов, метрики которых необходимо собирать:

- ❑ журнальные записи узлов;
- ❑ журнальные записи управляющего уровня Kubernetes:
  - API-сервер;
  - менеджера контроллеров;
  - планировщика;
- ❑ журнальные записи системы аудита Kubernetes;
- ❑ журнальные записи контейнеров с приложениями.

В случае с узлами имеет смысл собирать информацию о событиях, происходящих в основных сервисах. Например, вам могут пригодиться журнальные

записи демонов Docker, запущенных на рабочих узлах. Корректная работа данного демона необходима для корректной работы контейнеров. Сбор этих записей поможет вам диагностировать любые проблемы, которые могут возникнуть в работе системы Docker, и разобраться в любых внутренних неполадках ее демона. На узлах есть и другие ключевые сервисы, требующие ведения журнала.

Управляющий уровень Kubernetes состоит из нескольких компонентов, журнальные записи которых позволят вам лучше ориентироваться в его внутренних проблемах. Он является основой работоспособного кластера, поэтому вам следует собирать журналы, которые он хранит на своем сервере в файлах `/var/log/kube-APIserver.log`, `/var/log/kube-scheduler.log` и `/var/log/kube-controller-manager.log`. Менеджер контроллеров отвечает за создание объектов, определенных конечным пользователем. Представьте, к примеру, что вы как пользователь создали сервис типа `LoadBalancer`, который вошел в состояние ожидания и больше ничего не делает; при этом события Kubernetes не предоставляют всех подробностей для диагностики данной проблемы. Чтобы лучше разобраться в этой ситуации и быстрее диагностировать неполадку, журнальные записи необходимо собирать централизованным образом.

Журнальные записи системы аудита Kubernetes относятся к мониторингу безопасности, поскольку позволяют понять, кто и что делает в системе. Они могут содержать много лишней информации, вследствие чего лучше адаптировать их для своей среды. Часто на этапе начальной инициализации записи приводят к огромным скачкам в системе журналирования, поэтому обязательно ознакомьтесь с документацией по мониторингу журналов аудита в Kubernetes.

Журнальные записи контейнеров содержат ввод, генерируемый вашими приложениями. Их можно направлять в центральный репозиторий несколькими способами. Рекомендуемый подход состоит в том, чтобы выводить все журнальные записи приложений в STDOUT, поскольку это позволяет унифицировать журналирование и дает возможность системе мониторинга собирать их напрямую из демона Docker. Еще один вариант — использование паттерна проектирования *«Прицеп»* (Sidecar) и запуск контейнера для перенаправления журнальных записей в одном pod с контейнером приложения. Вам, вероятно, придется использовать этот паттерн, если ваше приложение хранит свой журнал в файловой системе.

Для управления журналами аудита в Kubernetes существует множество способов и конфигураций. Эти журналы могут содержать очень много лишней информации, вследствие чего хранение всех их записей может быть довольно затратным. Почитайте документацию по журналам аудита (https://oreil.ly/L84dM), чтобы адаптировать их для своей среды.

Инструменты для ведения журнала

Как и в случае с метриками, для сбора журнальных записей из Kubernetes и приложений, запущенных в кластере, существует огромное количество инструментов. Вы уже можете иметь какие-то из них, но вам следует понимать, как именно в них реализовано журналирование. Инструмент должен уметь работать в качестве Kubernetes DaemonSet и предоставлять вспомогательный контейнер для приложений, которые не направляют свои журнальные записи в STDOUT. Применение существующих решений имеет свои преимущества, поскольку у вас уже есть опыт их использования.

Ниже перечислены некоторые популярные инструменты, поддерживающие интеграцию с Kubernetes:

- Elastic Stack;
- Datadog;
- Sumo Logic;
- Sysdig;
- сервисы облачных провайдеров (GCP Stackdriver, Azure Monitor for containers и Amazon CloudWatch).

Если вы ищете средство для централизации журналов и не хотите лишних операционных расходов, то отличным выбором может стать SaaS-решение. На первый взгляд собственная система журналирования может показаться хорошей идеей, но по мере развития вашей среды на ее обслуживание будет уходить все больше и больше времени.

Журналирование с использованием стека EFK

В этой книге для мониторинга нашего кластера мы используем стек EFK (Elasticsearch, Fluentd и Kibana). Его реализация может стать хорошей отправной точкой, но рано или поздно вы, скорее всего, зададитесь вопросом: «А стоит ли тратить усилия на обслуживание собственной платформы жур-

налирования?» Ответ обычно отрицательный. Самостоятельное размещение средств ведения журнала может показаться хорошей идеей в самом начале, но через некоторое время станет слишком сложным. Операционная сложность подобных систем повышается вместе с масштабом вашей среды. У данной проблемы нет какого-то одного правильного решения, поэтому подумайте о том, диктуется ли самостоятельное размещение бизнес-требованиями. Кроме того, на основе стека EFK существует целый ряд SaaS-решений, так что если вы больше не захотите заниматься его обслуживанием, то сможете довольно легко перейти на внешний сервис.

Чтобы организовать систему мониторинга, следует развернуть такие компоненты:

- Elasticsearch Operator;
- Fluentd (направляет журнальные записи из среды Kubernetes в Elasticsearch);
- Kibana (средство визуализации с возможностью поиска, просмотра и взаимодействия с журнальными записями, хранящимися в Elasticsearch).

Разверните следующий манифест в своем кластере Kubernetes:

```
kubectl create namespace logging

kubectl apply -f https://raw.githubusercontent.com/dstrebel/kbp/master/
elasticsearch-operator.yaml -n logging
```

Разверните Elasticsearch Operator, чтобы агрегировать все передаваемые журнальные записи:

```
kubectl apply -f https://raw.githubusercontent.com/dstrebel/kbp/master/efk.yaml
-n logging
```

Эта команда развернет Fluentd и Kibana, что позволит нам направлять журнальные записи в Elasticsearch и визуализировать их с помощью Kibana.

В вашем кластере должны появиться такие pod:

```
kubectl get pods -n logging

efk-kibana-854786485-knhl5              1/1     Running   0       4m
elasticsearch-operator-5647dc6cb-tc2st  1/1     Running   0       5m
elasticsearch-operator-sysctl-ktvk9     1/1     Running   0       5m
elasticsearch-operator-sysctl-lf2zs     1/1     Running   0       5m
elasticsearch-operator-sysctl-r8qhb     1/1     Running   0       5m
es-client-efk-cluster-9f4cc859-sdrsl    1/1     Running   0       4m
es-data-efk-cluster-default-0           1/1     Running   0       4m
es-master-efk-cluster-default-0         1/1     Running   0       4m
fluent-bit-4kxdl                        1/1     Running   0       4m
fluent-bit-tmqjb                        1/1     Running   0       4m
fluent-bit-w6fs5                        1/1     Running   0       4m
```

Когда все pod станут активными (`Running`), подключитесь к серверу Kibana, перенаправив его порт к своей локальной системе:

```
export POD_NAME=$(kubectl get pods --namespace logging -l
"app=kibana,release=efk" -o jsonpath="{.items[0].metadata.name}")

kubectl port-forward $POD_NAME 5601:5601
```

Теперь откройте в своем браузере информационную панель Kibana по адресу http://localhost:5601.

Чтобы иметь возможность работать с журнальными записями, полученными из нашего кластера Kubernetes, сначала нужно создать индекс.

При первом запуске Kibana следует перейти на вкладку Management (Управление) и создать шаблон индексации журнальных записей Kubernetes. Система поможет вам пройти через все этапы данной процедуры.

Создав индекс, вы сможете выполнять поиск по учетным записям, используя синтаксис запросов Lucene. Например:

```
log:(WARN|INFO|ERROR|FATAL)
```

Этот запрос возвращает все записи, содержащие поля `fields` `warn`, `info`, `error` или `fatal`. Вы можете видеть результат на рис. 3.4.

Рис. 3.4. Информационная панель Kibana

Kibana поддерживает динамические запросы, поэтому вы можете создавать разные информационные панели для отслеживания своей среды.

Уделите некоторое время изучению различных журналов, которые можно визуализировать в Kibana.

Уведомления

Система уведомлений — палка о двух концах. Вам необходимо соблюдать баланс между тем, о чем следует уведомлять, и тем, что нужно просто отслеживать. Слишком большое количество уведомлений притупляет бдительность, в результате чего важные события могут затеряться в информационном шуме. Примером этого может послужить генерация уведомлений при каждом сбое pod. Вы могли бы подумать: «Почему бы мне не отслеживать сбои pod?» Но прелесть платформы Kubernetes в том, что она предоставляет средства для автоматической проверки работоспособности и перезапуска контейнеров. Ваши уведомления должны касаться тех событий, которые влияют на ваши цели уровня обслуживания (Service-Level Objectives, SLO). SLO — это определенные характеристики, поддающиеся измерению и согласованные с вашим конечным пользователем: например, доступность, пропускная способность, частота и время ответа. SLO позволяет определиться с тем, чего следует ожидать от вашей системы, и прояснить то, как она должна себя вести. Не располагая этими характеристиками, пользователь может составить собственное мнение о вашем сервисе, основываясь на нереалистичных ожиданиях. Уведомления в таких системах, как Kubernetes, требуют совершенно нового подхода, который отличается от всего того, к чему мы привыкли; нам следует сосредоточиться на том, как наш сервис воспринимают конечные пользователи. Например, если согласно SLO время ответа сервиса не должно быть больше 20 мс, но наблюдаемая вами латентность превышает средний показатель, то вас должны уведомить о проблеме.

Вам нужно определиться с тем, какие ситуации требуют вмешательства. Обычно речь идет о высокой нагрузке на процессор, повышенном использовании памяти или зависших процессах. На первый взгляд, это хорошие кандидаты для отправки уведомлений, но в реальности такие проблемы, скорее всего, не требуют принятия срочных мер или вызова дежурного инженера. Последнего следует беспокоить только в случаях, не решаемых без ручного вмешательства и влияющих на пользователей приложения. Хороший пример ситуации, когда уведомление не стоит направлять дежурному инженеру, — сценарий, в котором «проблема разрешается сама собой».

Если уведомление не требует принятия срочных мер, то вам лучше предусмотреть механизм, который автоматизирует исправление причин проблемы. Например, на случай заполнения диска можно автоматизировать процесс

удаления журнальных файлов. Кроме того, проведение *проверок работоспособности* (liveness probes) в ходе развертывания может помочь с автоматическим исправлением проблем, связанных с зависшими процессами приложения.

При конфигурации уведомлений необходимо продумать *время ожидания перед отправкой*. Если сделать его слишком коротким, то можно получить много ложных срабатываний. В целом во избежание этого время ожидания должно быть не меньше 5 мин. Чтобы не устанавливать данный показатель для каждой отдельной метрики, стоит подобрать стандартное значение по умолчанию. Например, вы можете руководствоваться методом постепенного увеличения времени ожидания: 5, 10, 30 мин, 1 ч и т. д.

При формировании сообщений (отправляющихся работникам компании) на основе уведомлений нужно позаботиться о предоставлении полезной информации. Например, вы можете указать ссылку на инструкции или другие сведения, которые должны помочь в устранении проблемы. В сообщение также следует включить информацию о вычислительном центре, регионе, владельце приложения и пострадавшей системе. Это позволит инженерам быстро выработать теорию о причинах проблемы.

Вам также нужно предусмотреть каналы сообщений для перенаправления срабатывающих уведомлений. Вы должны подумать над тем, кому именно следует послать сообщение, а не просто использовать общий список или адреса электронной почты ваших коллег. Если передавать уведомления большим группам людей, то такие уведомления рано или поздно начнут игнорировать и воспринимать как информационный шум. Сообщения должны попадать тем пользователям, которым следует взять на себя ответственность за решение проблемы. Систему уведомлений нельзя сделать идеальной с первой попытки, можно даже сказать, ее никогда не удастся довести до идеала. Тем не менее вы должны постоянно работать над ее улучшением, чтобы ваши сообщения были информативными и полезными. В противном случае вы будете зря отвлекать своих коллег, что может привести к проблемам в ваших системах.

Если хотите узнать больше о разных подходах к созданию уведомлений и управлению системами, то почитайте статью My Philosophy on Alerting[1] Роба Иващука (https://oreil.ly/YPxju) — она основана на его наблюдениях в ходе работы инженером по мониторингу (site reliability engineer, SRE) в Google.

[1] См. также: *Бейер Б., Джоунс К., Петофф Д., Мёрфи Н. Р.* Site Reliability Engineering. Надежность и безотказность как в Google. — СПб.: Питер, 2021. — С. 96.

Рекомендации по мониторингу, журналированию и созданию уведомлений

Ниже перечислены рекомендации относительно мониторинга, журналирования и создания уведомлений.

Мониторинг

- ❑ Отслеживайте степень использования, загруженности и частоту возникновения ошибок для узлов и всех компонентов Kubernetes. В случае с приложениями отслеживайте частоту и продолжительность запросов, а также частоту ошибок.
- ❑ Задействуйте мониторинг по принципу черного ящика для отслеживания симптомов и показателей работоспособности системы, которые нельзя спрогнозировать.
- ❑ Применяйте мониторинг по принципу белого ящика для исследования системы и ее внутренностей с помощью инструментирования.
- ❑ Создавайте метрики на основе временных рядов, чтобы получать высокоточные показатели. Это позволит лучше понять поведение вашего приложения.
- ❑ Используйте системы мониторинга наподобие Prometheus, которые позволяют формировать многоуровневые наборы метрик с помощью меток; так вы сможете получать лучшие сигналы о симптомах возникающих проблем.
- ❑ Задействуйте средние показатели для визуализации промежуточных значений и метрик, основанных на фактических данных. Применяйте суммирование, чтобы изобразить распределение по заданной метрике.

Журналирование

- ❑ Вы должны использовать журналирование в сочетании с мониторингом метрик для получения полноценной картины происходящего в вашей среде.
- ❑ Осторожно подходите к хранению журнальных записей дольше чем 30–45 дней. Если вам это действительно нужно, то используйте более дешевые ресурсы для долгосрочной архивации.

- Ограничивайте применение вспомогательных контейнеров для перенаправления журнальных записей по мере того, как они начинают потреблять слишком много ресурсов. Помещайте такие контейнеры в DaemonSet и выводите журнальные записи в STDOUT.

Создание уведомлений

- Относитесь с осторожностью к слишком частой рассылке уведомлений, поскольку это может плохо влиять на поведение процессов и людей.
- Всегда пытайтесь постепенно улучшать систему уведомлений и смиритесь с тем, что она никогда не будет идеальной.
- Уведомляйте о симптомах, влияющих на SLO и клиентов, но не о временных проблемах, которые не требуют немедленной реакции со стороны людей.

Резюме

В данной главе мы обсудили паттерны проектирования, методики и инструменты, пригодные для мониторинга систем с помощью сбора метрик и журнальных записей. Самый главный урок, который можно извлечь из представленного материала, состоит в том, что вам следует пересмотреть свой подход к мониторингу. И это необходимо делать с самого начала, иначе, как видно в многочисленных примерах, вы можете что-то упустить в работе системы. Мониторинг нужен для того, чтобы вы лучше разбирались в своей системе и могли сделать ее более отказоустойчивой; а это, в свою очередь, улучшает восприятие вашего приложения конечными пользователями. Мониторинг распределенных приложений и систем, таких как Kubernetes, требует много усилий, и вы должны быть изначально готовы к этому.

ГЛАВА 4

Конфигурация, Secrets и RBAC

Контейнеры компонуемы по своей природе, поэтому те, кто ими управляет, могут передавать им конфигурационные данные на этапе выполнения. Это позволяет отделить возможности приложения от среды, в которой оно работает: либо с помощью переменных среды, либо путем динамического подключения к контейнеру внешних томов. В результате можно легко менять конфигурацию приложения во время его запуска. Вам, как разработчику, необходимо учитывать динамическую природу данного поведения; вы должны предусмотреть использование переменных среды или чтение конфигурационных данных из определенного места, доступ к которому имеет пользователь приложения.

При перемещении конфиденциальных данных, например секретных ключей, в объект API Kubernetes необходимо понимать, как платформа защищает доступ к этому API. Самый распространенный механизм безопасности в Kubernetes — управление доступом на основе ролей (Role-Based Access Control, RBAC), на базе которого реализуется гибкая структура прав доступа. Механизм позволяет определять, какие вызовы API могут делать определенные пользователи или группы. В данной главе представлено краткое введение в RBAC, а также приведены некоторые рекомендации относительно этой системы.

Конфигурация с использованием объектов ConfigMap и Secret

Kubernetes имеет стандартные механизмы для предоставления приложению конфигурационной информации с помощью ресурсов `ConfigMap` и `Secret`. Основное различие между ними связано с тем, каким образом pod сохраняет полученные данные и как именно они хранятся в etcd.

Объекты ConfigMap

Очень часто приложения используют конфигурационную информацию через некий механизм, такой как аргументы командной строки, переменные среды или файлы, доступные в системе. Контейнеры позволяют разработчикам отделить эту информацию от кода, что дает возможность делать приложения по-настоящему портируемыми. API `ConfigMap` поддерживает внедрение пользовательской конфигурации. Эти объекты легко адаптируются под требования приложения и могут предоставлять конфигурацию в виде пар «ключ — значение», сложных массивов данных, таких как JSON и XML, или в проприетарном формате.

`ConfigMap` предоставляет конфигурационную информацию не только для pod, но и для более сложных системных сервисов, таких как контроллеры, пользовательские ресурсы, операторы и т. д. Как уже упоминалось ранее, API `ConfigMap` в основном рассчитан на обычные строковые значения. Для работы с более чувствительными данными лучше применять API `Secrets`.

Для получения доступа к данным, хранящимся в `ConfigMap`, можно использовать переменные среды или том, подключенный к pod.

Объекты Secret

Многие свойства и преимущества `ConfigMap` применимы и к объектам `Secret`. Основные различия состоят в фундаментальной природе последних: секретные данные должны храниться и обрабатываться так, чтобы их можно было легко скрывать и, возможно, шифровать во время хранения, если среда сконфигурирована соответствующим образом. Очень важно понимать, что кодировка base64, в которой хранятся секретные данные, не имеет ничего общего с шифрованием. Pod, в который внедрили объект `Secret`, может читать его содержимое в виде обычного текста.

Объекты `Secret` в Kubernetes рассчитаны на небольшие объемы данных, а их размер по умолчанию не может превышать 1 Мбайт. А учитывая, что они кодируются в base64, максимальный размер полезной информации равен примерно 750 Кбайт. Kubernetes поддерживает три типа объектов `Secret`:

- `generic` — это, как правило, обычные пары «ключ — значение», сформированные на основе файла, каталога или строковых литералов, передаваемых с помощью параметра `--from-literal=`, как показано ниже:

```
kubectl create secret generic mysecret --from-literal=key1=$3cr3t1
--from-literal=key2=@3cr3t2`
```

- `docker-registry` — используется утилитой kubelet при передаче в шаблон pod, если учетные данные, необходимые для аутентификации в закрытом реестре Docker, указаны в поле `imagePullsecret`:

```
kubectl create secret docker-registry registryKey --docker-server
myreg.azurecr.io --docker-username myreg --docker-password $up3r
$3cr3tP@ssw0rd --docker-email ignore@dummy.com
```

- `tls` — берет корректную пару открытых/закрытых ключей и создает на их основе объект `Secret` с TLS-шифрованием (при условии, что сертификат имеет корректный формат PEM). Затем этот объект можно передать в pod для установления SSL/TLS-соединений:

```
kubectl create secret tls www-tls --key=./path_to_key/wwwtls.key --
cert=./path_to_crt/wwwtls.crt
```

Объекты `Secret` можно подключать к pod через файловую систему tmpfs узла и затем отключать их, если соответствующего pod больше нет. Благодаря этому на диске узла не остается никаких лишних секретных данных. На первый взгляд схема выглядит безопасной. Однако необходимо понимать, что объекты `Secret` по умолчанию хранятся в базе данных etcd в виде обычного текста, поэтому важно, чтобы системные администраторы или поставщики облачных услуг принимали меры по защите среды etcd: устанавливали mTLS-соединения между узлами etcd, включали пассивное шифрование для хранимых данных и т. д. Последние версии Kubernetes используют хранилище etcd3 и позволяют использовать его стандартные средства шифрования. Но это требует ручного вмешательства в конфигурацию API-сервера — в частности, для корректного шифрования содержимого etcd нужно указать провайдера и подходящий накопитель для ключей. На момент выхода Kubernetes v1.10 (и в бета-версии v1.12) доступен провайдер KMS, который, как утверждается, обеспечивает более безопасное хранение ключей, делегируя этот процесс сторонним KMS-системам.

Общепринятые рекомендации по работе с API ConfigMap и Secret

Большая часть проблем, возникающих при использовании объектов `ConfigMap` и `Secret`, связана с ошибочными предположениями о том, каким образом обрабатываются изменения при обновлении хранящихся в этих объектах данных. Избежать неприятностей поможет понимание некоторых

общепринятых правил; а соблюдать их легче с помощью нескольких приемов, которые мы покажем ниже.

- Чтобы ваше приложение поддерживало динамические изменения без развертывания новых версий pod, подключайте объекты `ConfigMap`/`Secret` в качестве томов и используйте в своем коде механизм отслеживания файлов; это позволит обнаруживать измененные настройки и при необходимости автоматически обновлять конфигурацию. В следующем листинге показан объект `Deployment`, который подключает `ConfigMap` и файл `Secret` в виде тома:

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: nginx-http-config
    namespace: myapp-prod
data:
  config: |
    http {
      server {
        location / {
        root /data/html;
        }

        location /images/ {
          root /data;
        }
      }
    }

apiVersion: v1
kind: Secret
metadata:
  name: myapp-api-key
type: Opaque
data:
  myapikey: YWRtd5thSaW4=

apiVersion: apps/v1
kind: Deployment
metadata:
  name: mywebapp
  namespace: myapp-prod
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 8080
```

```
    volumeMounts:
    - mountPath: /etc/nginx
      name: nginx-config
    - mountPath: /usr/var/nginx/html/keys
      name: api-key
  volumes:
    - name: nginx-config
      configMap:
        name: nginx-http-config
        items:
        - key: config
          path: nginx.conf
    - name: api-key
      secret:
        name: myapp-api-key
        secretname: myapikey
```

При использовании поля volumeMounts следует учитывать несколько вещей. Во-первых, сразу после создания объектов ConfigMap/Secret их нужно указать в виде тома в спецификации pod и затем подключить этот том к файловой системе контейнера. Имя каждого ключа в ConfigMap/Secret превратится в отдельный файл в подключенном каталоге, а его значение станет содержимым данного файла. Во-вторых, не стоит подключать объекты ConfigMap/Secret с помощью свойства volumeMounts.subPath, иначе данные в томе не будут динамически обновляться при изменении конфигурации или секретных данных.

- Объекты `ConfigMap`/`Secret` должны быть доступны в пространстве имен pod, который их потребляет, еще до его создания. Для принудительного запуска pod даже в случае отсутствия `ConfigMap`/`Secret` можно использовать дополнительный флаг.
- Применяйте Admission-контроллер для получения нужной вам конфигурации и предотвращения развертывания объектов, у которых отсутствуют определенные конфигурационные значения. Например, вы можете требовать, чтобы во всех промышленных средах с рабочими заданиями на Java были заданы определенные свойства JVM. На сегодня существует альфа-версия API под названием `PodPresets`, который позволяет применять объекты `ConfigMap` и `Secret` ко всем pod с подходящими аннотациями, не прибегая к необходимости создавать собственный Admission-контроллер.
- Если приложения в вашей среде развертываются с помощью Helm, то вы можете использовать хук жизненного цикла (life cycle hook), чтобы

гарантировать развертывание шаблона `ConfigMap`/`Secret` до применения объекта `Deployment`.

- Некоторые приложения требуют, чтобы их конфигурация применялась в виде единого файла, такого как JSON или XML. В объекты `ConfigMap`/`Secret` можно записывать целые блоки текстовых данных, используя символ |. Например:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-file
data:
  config: |
    {
      "iotDevice": {
        "name": "remoteValve",
        "username": "CC:22:3D:E3:CE:30",
        "port": 51826,
        "pin": "031-45-154"
      }
    }
```

- Если приложение использует системные переменные среды для определения своей конфигурации, то вы можете привязать их к данным `ConfigMap`. Для этого есть два основных способа: подключить каждую пару «ключ — значение» в `ConfigMap` к pod в виде набора переменных среды, указав `envFrom`, а затем используя `configMapRef` или `secretRef`, либо же назначить отдельные ключи с их значениями с помощью `configMapKeyRef` или `secretKeyRef`.

- В случае использования метода с `configMapKeyRef` или `secretKeyRef` имейте в виду: если заданный ключ не существует, то pod не запустится.

- Если вы загружаете все пары «ключ — значение» из `ConfigMap`/`Secret` в pod с помощью `envFrom`, то любые ключи, значения которых являются некорректными переменными среды, будут пропущены; однако pod будет позволено запуститься. При этом сгенерируются событие типа `InvalidVariableNames` и соответствующее сообщение о пропущенных ключах. В следующем листинге показан пример объекта `Deployment` со ссылками на `ConfigMap` и `Secret` в качестве переменных среды:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-config
```

```
data:
  mysqldb: myappdb1
  user: mysqluser1

apiVersion: v1
kind: Secret
metadata:
  name: mysql-secret
type: Opaque
data:
  rootpassword: YWRtJasdhaW4=
  userpassword: MWYyZDigKJGUyfgKJBmU2N2Rm

apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-db-deploy
spec:
  selector:
    matchLabels:
      app: myapp-db
  template:
    metadata:
      labels:
        app: myapp-db
    spec:
      containers:
      - name: myapp-db-instance
        image: mysql
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 3306
        env:
          - name: MYSQL_ROOT_PASSWORD
            valueFrom:
              secretKeyRef:
                name: mysql-secret
                key: rootpassword
          - name: MYSQL_PASSWORD
            valueFrom:
              secretKeyRef:
                name: mysql-secret
                key: userpassword
          - name: MYSQL_USER
            valueFrom:
              configMapKeyRef:
                name: mysql-config
                key: user
```

```
        - name: MYSQL_DB
          valueFrom:
            configMapKeyRef:
              name: mysql-config
              key: mysqldb
```

- Если вам нужно передать в свой контейнер аргументы командной строки, то значения переменных среды можно получить с помощью интерполяции, используя синтаксис `$(ENV_KEY)`:

```
[...]
spec:
  containers:
  - name: load-gen
    image: busybox
    command: ["/bin/sh"]
args: ["-c", "while true; do curl $(WEB_UI_URL); sleep 10;done"]
    ports:
    - containerPort: 8080
    env:
    - name: WEB_UI_URL
      valueFrom:
        configMapKeyRef:
          name: load-gen-config
          key: url
```

- При использовании объектов `ConfigMap`/`Secret` в виде переменных среды необходимо понимать: изменения, вносимые в эти объекты, *не* приводят к обновлению значений в pod; чтобы изменения стали доступными, pod нужно перезапустить. Это можно сделать двумя способами: удалить pod и позволить контроллеру `ReplicaSet` его заменить или инициировать обновление объекта `Deployment`, который подчиняется стратегии обновления приложения, заданной в спецификации `Deployment`.

- Проще исходить из того, что изменение содержимого `ConfigMap`/`Secret` требует обновления всего объекта `Deployment`; благодаря этому ваш код всегда сможет подхватить новые конфигурационные данные, даже если вы применяете переменные среды или тома. Упростить этот процесс можно с помощью конвейера CI/CD: обновите свойство `name` объекта `ConfigMap`/`Secret` и ссылку в объекте `Deployment`, что автоматически повлечет за собой обновление вашего развертывания в рамках одной из стандартных стратегий обновления Kubernetes. Этот подход показан в следующем примере кода. Если вы развертываете свое приложение в Kubernetes с помощью Helm, то можете воспользоваться аннотацией в шаблоне `Deployment`, чтобы проверить контрольную сумму `sha256` объекта `ConfigMap`/`Secret`. И из-

менение содержимого `ConfigMap`/`Secret` заставит Helm обновить объект `Deployment` с помощью команды `helm upgrade`:

```
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  template:
    metadata:
      annotations:
        checksum/config: {{ include (print $.Template.BasePath "/config
        map.yaml") . | sha256sum }}
[...]
```

Рекомендации относительно разных видов секретных данных

Учитывая специфику секретных данных в API `Secrets`, вполне естественно, что для защиты самих данных существуют более подробные рекомендации.

- В исходной спецификации API `Secrets` описывается архитектура подключаемых модулей, которая позволяет конфигурировать хранение секретных данных на основе предъявляемых требований. Такие решения, как HashiCorp Vault, Aqua Security, Twistlock, AWS Secrets Manager, Google Cloud KMS и Azure Key Vault поддерживают внешние системы хранения секретных данных с повышенным уровнем шифрования и аудитоспособности по сравнению со стандартными средствами Kubernetes.

- Вы можете назначить `imagePullSecrets` учетной записи `serviceaccount`, которую pod будет использовать для автоматического подключения секретных данных; при этом его не нужно объявлять в `pod.spec`. Вы можете адаптировать служебную учетную запись под пространство имен своего приложения и напрямую добавить в нее `imagePullSecrets`. В результате она будет автоматически добавлена во все pod данного пространства имен:

  ```
  Create the docker-registry secret first
  kubectl create secret docker-registry registryKey --docker-server
  myreg.azurecr.io --docker-username myreg --docker-password $up3r$3cr3tP@ssw0rd
  --docker-email ignore@dummy.com
  ```

  Патч `serviceaccount` по умолчанию в пространстве имен, которое вы хотите настроить:

  ```
  kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name":
  "registryKey"}]}'
  ```

- Используйте возможности CI/CD для извлечения секретных данных из безопасного или зашифрованного хранилища с помощью аппаратного

модуля безопасности (Hardware Security Module, HSM) в процессе релиза кода. Это позволит разделить обязанности. Команда обеспечения безопасности может заняться созданием и шифрованием объектов `Secret`, в то время как разработчикам будет достаточно просто ссылаться на имена объектов, которые они ожидают. Это также соответствует рекомендуемым процедурам DevOps, делающим процесс доставки приложений более динамичным.

RBAC

При работе в крупных, распределенных средах очень часто требуется некий механизм безопасности, предотвращающий несанкционированный доступ к ключевым системам. Существует множество стратегий по ограничению доступа к ресурсам в вычислительных системах, но большинство из них состоят из одних и тех же этапов. Чтобы объяснить более наглядно процесс, происходящий в таких системах, как Kubernetes, воспользуемся жизненной аналогией: полетом за рубеж. Мы будем применять знакомые любому путешественнику понятия например паспорт, туристическая виза и таможенный или пограничный контроль.

1. Паспорт (аутентификация субъекта). Обычно для подтверждения личности пассажиру нужен паспорт, выданный неким правительственным органом. Это эквивалент учетной записи пользователя в Kubernetes. Для аутентификации пользователей Kubernetes применяет внешние системы сертификации, хотя управление служебными учетными записями происходит напрямую.

2. Виза, или режим пересечения границы (авторизация). Краткосрочное пребывание в стране гражданина с иностранным паспортом регламентируется официальным соглашением между странами (визой). Визы бывают разных типов и определяют, что приезжему гражданину позволено делать и как долго он может находиться в этой стране. Это эквивалент авторизации в Kubernetes. Платформа поддерживает разные методы авторизации, но самый распространенный из них — RBAC. Он позволяет гибко управлять доступом к разным возможностям API.

3. Таможенный или пограничный контроль (контроль доступа). При въезде в чужую страну обычно приходится иметь дело с неким органом власти, представители которого проверяют документы, включая паспорт и визу. Кроме того, ввозимый багаж очень часто проверяется на соответствие за-

конам данной страны. Это эквивалент контроллеров доступа в Kubernetes. Они могут принимать, отклонять и изменять запросы к API в зависимости от заданных правил и политик. Kubernetes предоставляет много встроенных контроллеров доступа, включая `PodSecurity`, `ResourceQuota` и `ServiceAccount`, и поддерживает динамический контроль доступа с помощью проверяющих (validating) и изменяющих (mutating) контроллеров (admission controllers).

Этот раздел посвящен наименее понятной и наиболее игнорируемой из перечисленных выше областей: RBAC. Прежде чем переходить к рекомендациям, следует познакомиться с основными понятиями данной системы.

Основные концепции RBAC

Процесс RBAC в Kubernetes состоит из трех главных компонентов, которые необходимо определить: субъекта, правила и привязки ролей.

Субъекты

Первый компонент — субъект — это то, что проверяется на возможность доступа. В качестве субъекта обычно выступает пользователь, служебная учетная запись или группа. Как уже упоминалось ранее, работа с пользователями, равно как и с группами, происходит за пределами Kubernetes, в одном из модулей авторизации. Эти модули поддерживают базовую HTTP-аутентификацию, клиентские сертификаты x.509 и токены (bearer tokens). Чаще всего применяются либо клиентские сертификаты x.509, либо некая разновидность токенов с использованием какой-нибудь системы OpenID Connect наподобие Azure Active Directory (Azure AD), Salesforce или Google.

Служебные учетные записи отличаются от пользовательских тем, что привязываются к пространствам имен, хранятся внутри Kubernetes и управляются стандартными контроллерами; они предназначены для доступа процессам, а не людям.

Правила

Говоря простым языком, это фактический список действий, которые можно выполнять с определенным объектом (ресурсом) или группой объектов в API. Действия соответствуют стандартным операциям CRUD (Create, Read, Update, Delete — «создать, прочитать, обновить, удалить»), но с некоторыми

дополнительными возможностями, характерными для Kubernetes: `watch`, `list` и `exec`. Объекты соответствуют разным API-компонентам и группируются по категориям. Pod-объекты, к примеру, являются частью основного API, и на них можно ссылаться с помощью поля `apiGroup: ""`; в то же время объекты `Deployment` принадлежат к API-группам `apps`. Это самая сильная сторона процесса RBAC, которая, вероятно, отпугивает и запутывает пользователей, пытающихся налаживать контроль доступа.

Роли

Роли позволяют определить область действия заданных правил. Kubernetes поддерживает два вида ролей: `role` и `clusterRole`; разница в том, что `role` относится к отдельному пространству имен, тогда как `clusterRole` действует на уровне целого кластера, во всех пространствах сразу. Ниже показан пример определения роли, ограниченной одним пространством имен:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-viewer
rules:
- apiGroups: [""] # "" указывает на основную API-группу
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

RoleBinding

`RoleBinding` позволяет привязывать субъекты, такие как пользователи или группы, к ролям. Привязка имеет два режима: `roleBinding`, который относится к отдельному пространству имен, и `clusterRoleBinding`, действующий на уровне всего кластера. Ниже представлен пример `RoleBinding` для пространства имен:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: noc-helpdesk-view
  namespace: default
subjects:
- kind: User
  name: helpdeskuser@example.com
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role       # здесь может быть только Role или ClusterRole
  name: pod-viewer # должно совпадать с именем роли (Role или ClusterRole),
                   # к которой мы привязываемся
  apiGroup: rbac.authorization.k8s.io
```

Рекомендации по работе с RBAC

RBAC — ключевой компонент в обеспечении безопасности, надежности и стабильности среды Kubernetes. Концепции, на которых основана эта система, могут показаться сложными, но часть основных трудностей можно преодолеть с помощью нескольких рекомендуемых методик.

- Приложения, разрабатываемые для работы в Kubernetes, редко нуждаются в ролях и привязках RBAC. Исключение составляют случаи, когда код приложения взаимодействует непосредственно с API Kubernetes.
- Если приложению в самом деле необходим прямой доступ к API Kubernetes (например, чтобы изменять конфигурацию в зависимости от того, какие конечные точки (endpoints) добавляются в сервис (`Service`)) или список всех pod в заданном пространстве имен, то лучше всего создать еще одну служебную учетную запись и указать ее в спецификации pod. Затем следует создать роль с минимальным набором привилегий, достаточным для выполнения поставленных перед ней задач.
- Используйте сервис OpenID Connect, поддерживающий идентификацию и при необходимости двухфакторную аутентификацию. Привязывайте группы пользователей к ролям с минимальным набором привилегий, достаточным для выполнения работы.
- Помимо упомянутых выше методик, следует также использовать системы доступа JIT (just in time — «со строгим ограничением по времени»), чтобы в систему могли войти инженеры по мониторингу (site reliability engineers, SRE), операторы и те, кому может понадобиться краткосрочное повышение привилегий в целях выполнения определенных задач. В качестве альтернативы для них можно выделять отдельные учетные записи с более строгим контролем и повышенными привилегиями, которые выдаются в виде роли, назначаемой пользователям или группе пользователей.
- Определенные служебные учетные записи должны использоваться для инструментов CI/CD, которые развертывают ваши кластеры Kubernetes. Это обеспечивает аудитоспособность в рамках кластера и позволяет понять, кто ответственен за развертывание или удаление каких-либо объектов.
- Если вы развертываете приложения с помощью Helm, то стандартная служебная учетная запись, Tiller, развертывается в `kube-system`. Вместо этого Tiller лучше развернуть в каждом пространстве имен, используя специально созданную служебную учетную запись, ограниченную этим

конкретным пространством. В инструменте CI/CD, который на подготовительном этапе вызывает команду Helm `install/upgrade`, инициализируйте клиент Helm с помощью служебной учетной записи и пространства имен, предназначенного для развертывания. Имя данной записи может быть одним и тем же для каждого пространства, но пространство для развертывания следует указать отдельно. Необходимо упомянуть, что на момент выхода этой книги Helm версии 3 находится в состоянии alpha и одной из его ключевых особенностей является то, что присутствие Tiller в кластере больше не требуется. Ниже показан пример процедуры инициализации Helm с помощью служебной учетной записи (`ServiceAccount`) и пространства имен:

```
kubectl create namespace myapp-prod

kubectl create serviceaccount tiller --namespace myapp-prod

cat  <<EOF | kubectl apply -f -
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: tiller
  namespace: myapp-prod
rules:
- apiGroups: ["", "batch", "extensions", "apps"]
  resources: ["*"]
  verbs: ["*"]
EOF

cat <<EOF | kubectl apply -f -
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: tiller-binding
  namespace: myapp-prod
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: myapp-prod
roleRef:
  kind: Role
  name: tiller
  apiGroup: rbac.authorization.k8s.io
  EOF

helm init --service-account=tiller --tiller-namespace=myapp-prod

helm install ./myChart --name myApp --namespace myapp-prod
--set global.namespace=myapp-prod
```

Отдельные публичные Helm-чарты не содержат поля для выбора пространства имен, в котором следует развертывать компонент приложения. Вследствие этого может потребоваться ручное редактирование чарта или использование учетной записи Tiller с повышенными привилегиями, чтобы получить права на развертывание в любом пространстве имен, а также на создание новых пространств.

- Ограничивайте доступ для любых приложений, которым нужно выполнять действия `watch` и `list` с API `Secrets`. Эти действия, в сущности, позволяют приложению или человеку, развернувшему pod, просматривать секретные данные в соответствующем пространстве имен. Если приложение использует этот API для доступа к конкретным объектам `Secret`, то позвольте ему выполнять действие `get` только для этих объектов (в дополнение к секретным данным, которые назначены ему напрямую).

Резюме

Принципы разработки приложений с поддержкой облачно-ориентированной доставки — тема для другой книги, но тот факт, что четкое разделение конфигурации и кода является залогом успеха, ни у кого не вызывает сомнений. Благодаря наличию в Kubernetes стандартных API `ConfigMap` (для обычных данных) и `Secrets` (для чувствительных данных) этот процесс теперь можно сделать декларативным. Стандартные средства Kubernetes позволяют представлять и хранить все больше и больше важной информации, поэтому крайне важно защитить доступ к ним с помощью правильно организованных процессов обеспечения безопасности, таких как RBAC и интегрированные системы аутентификации.

Как вы увидите дальше, эти принципы пронизывают каждый аспект развертывания сервисов на платформе Kubernetes, позволяя создавать стабильные, надежные, безопасные и устойчивые системы.

ГЛАВА 5

Непрерывная интеграция, тестирование и развертывание

В этой главе мы рассмотрим ключевые аспекты внедрения процесса непрерывной интеграции и непрерывного развертывания (continuous integration/continuous deployment, CI/CD) для доставки приложения в кластер Kubernetes. Построение тесно интегрированного процесса придаст вам уверенности при доставке приложений в промышленную среду. Мы рассмотрим методы, инструменты и процедуры, с помощью которых вы сможете наладить CI/CD в своих условиях. Задача CI/CD состоит в создании полностью автоматизированного процесса — от сохранения кода разработчиком до развертывания обновлений в промышленной среде. Ручного обновления приложений, развернутых в Kubernetes, лучше избегать, поскольку это может приводить к большому количеству ошибок, смещению конфигурации, ненадежным развертываниям и общей потере гибкости при доставке кода.

В этой главе мы обсудим следующие темы:

- управление версиями;
- непрерывную интеграцию;
- тестирование;
- создание тегов для образов;
- непрерывное развертывание;
- стратегии развертывания;
- тестирование развернутого кода;
- хаотическое тестирование.

Мы также пройдемся по примеру процесса CI/CD, состоящего из таких этапов, как:

- загрузка изменений в Git-репозиторий;
- сборка программного кода;

- тестирование кода;
- сборка образа контейнера в случае успешного прохождения тестов;
- загрузка образа контейнера в реестр;
- развертывание приложения в Kubernetes;
- тестирование развернутого приложения;
- обновление `Deployment` без простоя в работе.

Управление версиями

Каждый процесс CI/CD начинается с системы контроля версий, которая ведет историю изменений в коде и конфигурации приложения. Git — платформа для управления исходными текстами и отраслевым стандартом в этой области. Каждый Git-репозиторий содержит *главную ветку* (master branch), предназначенную для промышленного кода. У вас также будут другие ветки для общей разработки и создания новых возможностей, которые в конечном счете сольются с главной. Существует множество стратегий ветвления, выбор которых сильно зависит от организационной структуры и разделения обязанностей. Опыт показывает, что совместное хранение кода приложения и его конфигурации, такой как манифест Kubernetes или Helm-чарты, способствует общению и совместной работе внутри команды, которые являются важными принципами DevOps. Взаимодействие разработчиков и инженеров по эксплуатации в одном репозитории гарантирует доставку приложений в промышленную среду.

Непрерывная интеграция

Непрерывная интеграция (continuous integration, CI) — процесс, когда изменения кода постоянно сохраняются в репозитории системы контроля версий. Это подразумевает более частую фиксацию мелких изменений. Каждый раз в момент фиксации кода в репозитории запускается процесс сборки. Это ускоряет цикл обратной связи и позволяет сразу же увидеть проблемы, которые могут возникнуть. У вас может назреть справедливый вопрос: «Почему меня должно интересовать то, как собирается приложение, — разве за это не должен отвечать разработчик?» С традиционной точки зрения вы правы, но по мере того, как в компаниях начинает развиваться культура DevOps, команде инженеров по эксплуатации все чаще приходится иметь дело с кодом и рабочими процессами разработки ПО.

Существует множество решений, предоставляющих возможности CI, и одно из самых популярных — Jenkins.

Тестирование

Тесты выполняются для того, чтобы обеспечить быстрый цикл обратной связи на случай, если внесенные изменения нарушают процесс сборки. Выбор фреймворка для тестирования зависит от используемого языка. Например, чтобы выполнить набор модульных тестов для кодовой базы на языке Go, можно задействовать команду `go test`. Наличие обширного набора тестов помогает предотвратить доставку некачественного кода в промышленную среду. Если какой-то тест в наборе не был пройден, то процесс сборки следует считать неудачным. При наличии в вашей кодовой базе непройденных тестов собирать и загружать в реестр образ контейнера крайне нежелательно.

Но у вас опять может возникнуть вопрос: «Разве за создание тестов отвечает не разработчик?» Когда вы начнете автоматизировать доставку инфраструктуры и приложений в промышленную среду, вам нужно будет подумать об автоматическом тестировании всех элементов кодовой базы. Например, в главе 2 мы обсуждали, как с помощью Helm упаковывать приложения для Kubernetes. Helm поддерживает команду `helm lint`, которая выполняет ряд тестов для заданного чарта, пытаясь найти потенциальные проблемы. Полноценный процесс доставки кода требует выполнения множества разных тестов. За модульное тестирование приложений, к примеру, отвечают разработчики, но предварительные тесты — общая обязанность. Тестирование кодовой базы и ее доставка в промышленную среду — аспект командной работы, который должен быть реализован от начала и до конца.

Сборка контейнеров

Вы должны оптимизировать размер собираемых вами образов. Компактный образ более безопасен и требует меньше времени на загрузку и развертывание. Оптимизацию размера образа можно проводить разными способами, но некоторые из них имеют свои недостатки. Стратегии, указанные ниже, помогут вам собрать как можно меньший образ для своего приложения.

- *Многоэтапная сборка* (multitage builds). Позволяет избавиться от зависимостей, которые не требуются для выполнения вашего приложения. Например, если взять Golang, то нам не нужны все инструменты, исполь-

зующиеся для построения статического исполняемого файла, поэтому в файле Dockerfile можно описать процесс сборки, в результате которого итоговый образ будет содержать лишь двоичный файл приложения.

- *Базовые образы без операционной системы* (distroless base images). Позволяют избавиться от всех ненужных двоичных файлов и командных оболочек. Это существенно уменьшает размер образа и повышает безопасность. Но, с другой стороны, отсутствие командной оболочки не позволяет подключить к образу отладчик. На первый взгляд идея может показаться отличной, но при этом отладка приложений часто оказывается крайне неудобной. В состав таких образов не входят диспетчер пакетов, командная оболочка и другие компоненты операционной системы, поэтому вы можете не иметь доступа к привычным средствам отладки.
- *Оптимизированные базовые образы* — такие, основной акцент в которых сделан на компактности и удалении лишних слоев ОС. Например, Alpine предоставляет базовый образ с минимальным размером всего 10 Мбайт, который тем не менее позволяет подключать локальный отладчик. Другие дистрибутивы тоже обычно имеют оптимизированные версии, такие как Debian Slim. Это может оказаться хорошим вариантом, поскольку, помимо уменьшения размера и минимизации риска нарушения безопасности, вы получаете поддержку привычных вам средств разработки.

Оптимизация образов — чрезвычайно важный процесс, о котором пользователи часто забывают. Этому могут быть свои причины, такие как наличие корпоративного стандарта, в котором перечислены ОС, одобренные для применения в промышленных условиях. Но если вы хотите максимально повысить эффективность своих контейнеров, то с этим стоит побороться.

Как показывает наш опыт, компании, которые переходят на Kubernetes, вначале, как правило, успешно используют свою текущую ОС, но затем выбирают более оптимизированный образ, такой как Debian Slim. Познакомившись поближе с эксплуатацией и разработкой в контейнерной среде, вы сможете комфортно работать с образами, не содержащими операционной системы.

Назначение тегов образам контейнеров

Следующий этап в процессе CI — сборка образа Docker, который можно развернуть в среде выполнения. При этом необходимо предусмотреть стратегию назначения тегов, чтобы вы могли легко различать разные версии

развернутых вами образов. Одна из рекомендаций, важность которых сложно переоценить, звучит так: не используйте для своих образов тег `latest`. Он не указывает ни на какую *версию* и мешает разобраться в том, какие изменения вошли в развернутое приложение. Каждый образ, созданный в процессе CI, должен иметь уникальный тег.

Ниже перечислено несколько стратегий назначения тегов, эффективность которых подтверждается нашим опытом. С их помощью вы сможете легко определить изменения, внесенные в код, и сборку, к которой они относятся.

- *ID сборки*. Процессу сборки, который выполняется в рамках CI, назначается идентификатор. Если указать его в теге, то это позволит знать, в рамках какого процесса был создан образ.
- *Система сборки — ID сборки*. То же, что и предыдущий вариант, но с указанием системы сборки — на случай, если таких систем несколько.
- *Хеш Git*. При коммите нового кода Git генерирует хеш. Использование этого хеша в теге позволяет легко понять, на основе какого коммита был создан образ.
- *Хеш Git-ID сборки*. Это позволяет ссылаться как на коммит кода, так и на процесс сборки, в ходе которого был сгенерирован образ. Единственный недостаток данного варианта заключается в том, что тег может получиться довольно длинным.

Непрерывное развертывание

Непрерывное развертывание (continuous deployment, CD) — это когда изменения, успешно прошедшие через процесс CI, развертываются в промышленной среде без человеческого вмешательства. Для этого отлично подходят контейнеры. Образ контейнера — неизменяемый объект, который можно шаг за шагом провести из отладочной среды в промежуточную, а затем и в промышленную. Например, одна из основных проблем, с которыми мы постоянно сталкивались, заключалась в сопровождении согласованных сред. Почти все сталкивались с объектами `Deployment`, которые нормально работали в промежуточной среде, но ломались в промышленных условиях. Это связано со *смещением конфигурации* — ситуацией, когда компоненты используют различные библиотеки и версии в разных средах. Kubernetes позволяет описывать объекты `Deployment` декларативным способом, с поддержкой версий и доставкой в согласованной манере.

Следует отметить, что, прежде чем сосредотачиваться на CD, вы должны сначала организовать надежный процесс CI. Не имея обширного набора тестов для выявления проблем на ранних этапах, вы не сможете предотвратить развертывание некачественного кода во все свои среды.

Стратегии развертывания

Итак, мы усвоили основные принципы CD. Теперь рассмотрим разные стратегии развертывания, которые вы можете применять. Kubernetes предоставляет несколько встроенных механизмов развертывания новых версий приложения, хотя вы можете использовать и более продвинутые стратегии. Здесь мы исследуем следующие способы доставки обновлений:

- плавающие обновления (rolling updates);
- сине-зеленые развертывания (blue/green deployments);
- канареечные развертывания (canary deployments).

Плавающие обновления встроены в Kubernetes. Они позволяют обновлять активное приложение, не останавливая его работу. Например, если у вас есть клиентское приложение версии 1 и вы хотите перевести его объект развертывания на версию 2, то Kubernetes обновит его реплики «плавающим» способом. Этот процесс показан на рис. 5.1.

Объект `Deployment` также позволяет указать максимальное количество реплик, которые нужно обновить, и максимально допустимое количество pod во время развертывания. Ниже приведен пример манифеста со стратегией плавающих обновлений:

```
kind: Deployment
apiVersion: v1
metadata:
  name: frontend
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: frontend
        image: brendanburns/frontend:v1
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1       # максимальное количество одновременно обновляемых реплик
      maxUnavailable: 1 # максимально допустимое количество
                        # недоступных реплик во время развертывания
```

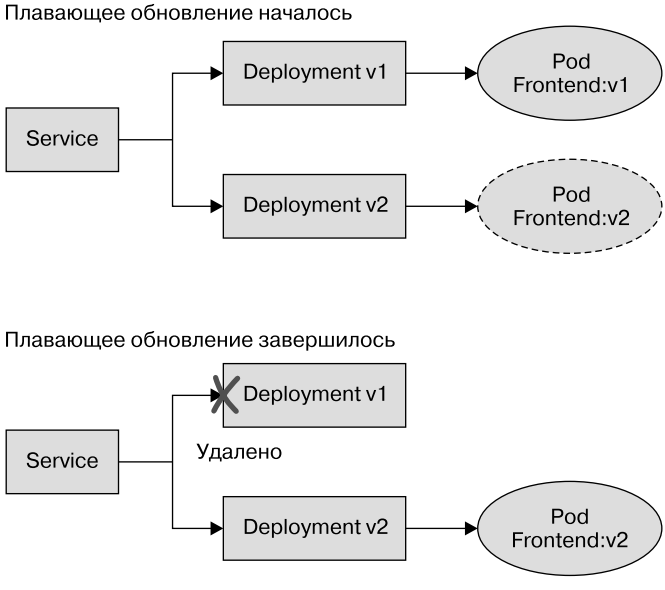


Рис. 5.1. Плавающее обновление в Kubernetes

К плавающим обновлениям следует подходить с осторожностью, поскольку эта стратегия может приводить к разрыву сетевых соединений. Чтобы бороться с этой проблемой, можно использовать *проверки готовности* и *preStop*-хук жизненного цикла контейнера. Первые позволяют убедиться в том, что новая развернутая версия готова обрабатывать трафик, а вторые дают возможность дождаться закрытия всех соединений с текущей версией приложения. Хук жизненного цикла вызывается перед завершением работы контейнера. Он должен успеть выполниться до получения сигнала о завершении процесса (`SIGTERM`). Проверка готовности и хук жизненного цикла реализованы в следующем примере:

```
kind: Deployment
apiVersion: v1
metadata:
  name: frontend
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: frontend
        image: brendanburns/frontend:v1
        livenessProbe:
```

```
        # ...
      readinessProbe:
        httpGet:
          path: /readiness # конечная точка для проверки
          port: 8888
      lifecycle:
        preStop:
          exec:
            command: ["/usr/sbin/nginx","-s","quit"]
strategy:
  # ...
```

В этом примере preStop-хук жизненного цикла контейнера корректно завершает процесс NGINX, тогда как сигнал `SIGTERM` просто прерывает его работу.

Еще одной проблемной особенностью плавающих обновлений является то, что в процессе развертывания одновременно работают две версии приложения. Схема вашей базы данных должна быть совместима с обеими версиями. Вы также можете задействовать стратегию ротации, согласно которой в схеме должны быть указаны новые столбцы, созданные новой версией приложения. По завершении плавающего обновления старые столбцы можно удалить.

Мы также определили в манифесте нашего `Deployment` проверки готовности (readiness probe) и работоспособности (liveness probe). Первая позволяет убедиться в том, что ваше приложение готово к обслуживанию трафика, прежде чем привязывать его к сервису в качестве конечной точки. Вторая дает возможность следить за тем, чтобы приложение работало как следует, и перезапускать проблемные pod. Автоматический перезапуск производится только в случае, если pod завершил работу с ошибкой. Например, при обнаружении в ходе проверки конечной точки зависшего pod, который не может завершить работу, Kubernetes перезапустит его.

Сине-зеленые развертывания позволяют выпускать приложения предсказуемым образом и определять, в какой момент трафик должен переключиться на новую среду. Это позволяет лучше контролировать развертывание новых версий. У вас должно быть достаточно ресурсов для развертывания новой среды в дополнение к существующей. Эта стратегия имеет множество преимуществ, например возможность легко откатиться на предыдущую версию приложения. Но у нее есть определенные аспекты, которые следует иметь в виду.

- Этот метод развертывания может затруднить миграцию базы данных, поскольку вам необходимо учитывать активные транзакции и совместимость изменений, вносимых в схему.

- Существует риск непреднамеренного удаления обеих сред.
- Вам нужны дополнительные ресурсы для работы сразу двух сред.
- Во время гибридного развертывания взаимодействие старой и новой версий может привести к проблемам.

Сине-зеленое развертывание изображено на рис. 5.2.

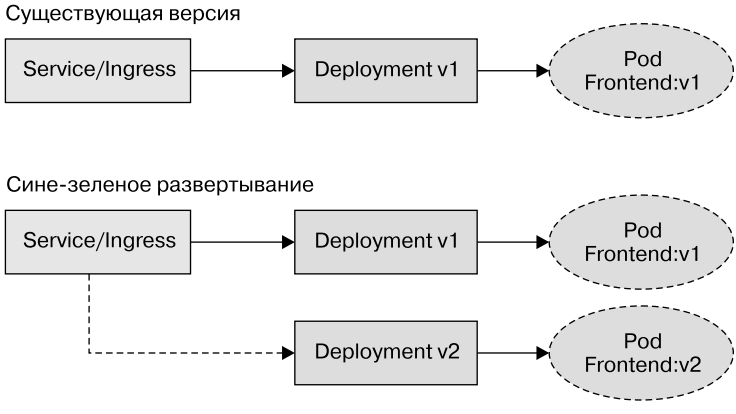


Рис. 5.2. Сине-зеленое развертывание

Канареечное развертывание очень похоже на сине-зеленое, но позволяет намного лучше контролировать перенаправление трафика к новой версии. Большинство современных контроллеров входящего трафика поддерживают частичное смещение нагрузки, но, кроме него, вы можете реализовать механизм межсервисного взаимодействия, такой как Istio, Linkerd или HashiCorp Consul, который предоставит вам целый ряд возможностей для воплощения этой стратегии развертывания.

Канареечное развертывание позволяет тестировать новые возможности среди ограниченного круга пользователей. Например, вы можете выкатить новую версию приложения и сделать так, чтобы она была доступна только для 10 % ваших посетителей. Следовательно, если у вас возникнут проблемы с некорректным `Deployment` или неработающими функциями, то они затронут лишь какое-то подмножество пользователей. При отсутствии ошибок вы сможете направлять все больше трафика к новой версии приложения. Канареечное развертывание позволяет применять и более продвинутые методики: например, вы можете выпустить новую версию в каком-то отдельном регионе

или только для пользователей с определенными характеристиками. Такой подход часто называют A/B или темным выпуском, поскольку пользователи не знают, что участвуют в тестировании новых возможностей.

Канареечные развертывания имеют следующие особенности, на которые следует обратить внимание (вдобавок к перечисленным для сине-зеленых развертываний). Вы должны:

- иметь возможность направлять трафик определенной доле пользователей;
- хорошо разбираться в поведении существующей версии, чтобы сравнить ее с новой;
- иметь доступ к метрикам, чтобы понимать, в каком состоянии находится новый релиз: в «хорошем» или «плохом».

Пример канареечного развертывания показан на рис. 5.3.

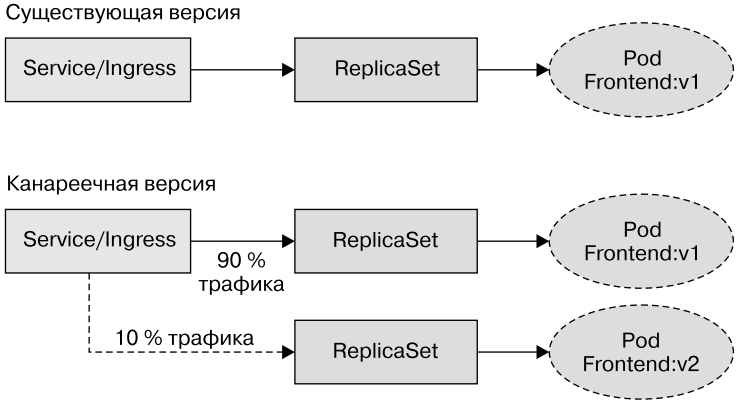


Рис. 5.3. Канареечное развертывание

Канареечным развертываниям тоже присущи проблемы с одновременным выполнением разных версий приложения. Схема вашей базы данных должна поддерживать обе версии. В ходе применения этих стратегий вам следует сосредоточиться на работе нескольких версий и на том, как поступать с сервисами, которые зависят от них. Вам необходимо предусмотреть стабильные контракты для API и позаботиться о том, чтобы ваши хранилища данных поддерживали все реплики, выполняющиеся параллельно.

Тестирование в промышленных условиях

Тестирование в промышленных условиях помогает удостовериться в том, что ваше приложение устойчиво, масштабируемо и корректно взаимодействует с пользователями. Однако при этом следует понимать, что, несмотря на потенциальные трудности и риски, *выполнение тестов в промышленной среде* стоит потраченных усилий и позволяет сделать ваши системы надежными. Но прежде, чем переходить к реализации, необходимо обсудить кое-какие важные моменты. Вы должны убедиться в том, что имеете комплексную стратегию наблюдаемости, которая позволит вам выявить последствия тестирования в промышленных условиях. Если вы не можете наблюдать за метриками, которые влияют на взаимодействие конечного пользователя с вашим приложением, то не будете иметь четкого представления о том, на чем следует сосредоточиться в целях повышения устойчивости системы. Вам также нужно позаботиться о высокой степени автоматизации, чтобы ваша система могла автоматически восстанавливаться после инициированных вами сбоев.

Для снижения риска и эффективного тестирования своих промышленных систем вам придется реализовать много разных инструментов. Часть из них мы уже обсуждали в данной главе, но есть и другие: например, распределенная трассировка, инструментирование, хаотическое тестирование и затенение трафика. Прибавьте к этому инструменты, которые мы уже упоминали:

- ❑ канареечные развертывания;
- ❑ A/B-тестирование;
- ❑ смещение трафика;
- ❑ флаги для переключения поведения функций.

Хаотическое тестирование было изобретено компанией Netflix. Это метод развертывания экспериментальных возможностей в промышленной системе с целью обнаружения ее слабых мест. Такое тестирование позволяет наблюдать за поведением системы в ходе контролируемого эксперимента. Ниже перечислены меры, которые необходимо принять, прежде чем начинать «живой» эксперимент.

1. Сформируйте четкое представление о «нормальном» состоянии системы.
2. Подготовьте разного рода «реальные» события, которые могут повлиять на систему.

3. Соберите контрольную группу и разработайте эксперимент.
4. Проведите эксперимент и сравните результаты с нормальным состоянием.

При подготовке экспериментов очень важно минимизировать потенциальные риски. Кроме того, эксперименты следует автоматизировать, поскольку их проведение может оказаться тяжелой работой.

К этому моменту у вас уже мог назреть вопрос: «Почему бы мне не выполнить тестирование в промежуточной среде?» Наш опыт говорит о том, что у этого подхода есть проблемы, включая следующие:

- расхождения в развертывании ресурсов;
- смещение конфигурации относительно промышленной среды;
- обычно трафик и поведение пользователей генерируются искусственным образом;
- количество сгенерированных запросов не отражает реальную нагрузку;
- в промежуточной среде может не хватать некоторых средств мониторинга;
- развернутые хранилища содержат не те данные и испытывают другие нагрузки по сравнению с промышленной средой.

Следует отдельно подчеркнуть: вы должны быть уверены в надежности своих средств мониторинга, поскольку эта методика имеет тенденцию к некорректности в системах с недостаточной наблюдаемостью. Кроме того, начинайте с небольших экспериментов; исследование их последствий придаст вам уверенности.

Подготовка процесса и проведение хаотического эксперимента

На первом этапе вы должны создать на сайте GitHub собственный репозиторий на основе уже готового (oreil.ly/TtJfd). Вы будете работать с ним на протяжении оставшейся части данной главы.

Подготовка CI

Вы уже познакомились с процессом CI. Теперь вам необходимо подготовить сборку кода, который вы клонировали выше.

В этом примере мы будем использовать сервис drone.io. Вам нужно создать бесплатную учетную запись (cloud.drone.io), используя учетные данные GitHub (в результате ваши репозитории будут зарегистрированы на сайте Drone и вы сможете их синхронизировать). Первым делом нужно создать в настройках несколько секретных значений, чтобы вы могли загрузить свое приложение в реестр Docker Hub и развернуть его в своем кластере Kubernetes.

Нажмите в своем репозитории Drone кнопку Settings (Настройки) и добавьте следующие секретные значения (рис. 5.4):

- `docker_username`;
- `docker_password`;
- `kubernetes_server`;
- `kubernetes_cert`;
- `kubernetes_token`.

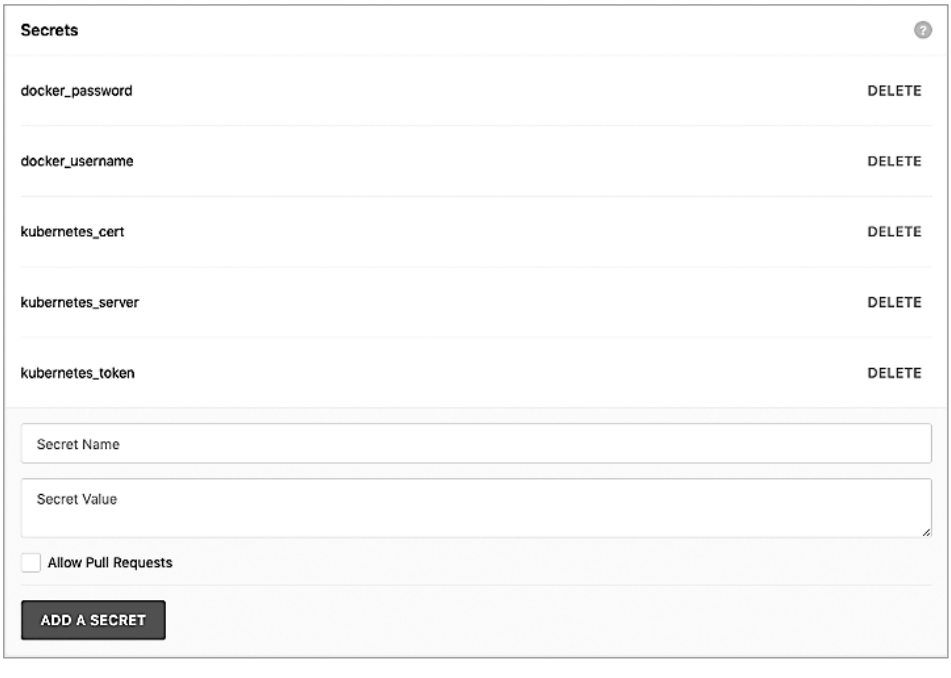


Рис. 5.4. Конфигурация секретных значений в Drone

Укажите имя пользователя и пароль, которые вы использовали при регистрации в Docker Hub. Ниже показано, как создать служебную учетную запись и сертификат для Kubernetes и получить токен.

У вашего сервера Kubernetes должен быть публично доступен API-сервер.

Для выполнения инструкций, приведенных в этом разделе, вам потребуются привилегии cluster-admin в вашем кластере Kubernetes.

Для получения конечной точки API можно использовать следующую команду:

```
kubectl cluster-info
```

Вы должны увидеть примерно такое сообщение: *Kubernetes master is running at https://kbp.centralus.azmk8s.io:443*. Сохраните указанный адрес в поле `kubernetes_server`.

Теперь создадим служебную учетную запись, с помощью которой Drone будет подключаться к кластеру. Выполните следующую команду:

```
kubectl create serviceaccount drone
```

Создадим привязку `clusterrolebinding` для `serviceaccount`:

```
kubectl create clusterrolebinding drone-admin \
  --clusterrole=cluster-admin \
  --serviceaccount=default:drone
```

Вслед за этим можно получить токен `serviceaccount`:

```
TOKENNAME=`kubectl -n default get serviceaccount/drone -o
jsonpath='{.secrets[0].name}'`
TOKEN=`kubectl -n default get secret $TOKENNAME -o jsonpath='{.data.token}' |
base64 -d`
echo $TOKEN
```

Вывод `echo $TOKEN` следует сохранить в поле `kubernetes_token`.

Для аутентификации в кластере вам также понадобится пользовательский сертификат, поэтому выполните следующую команду и вставьте содержимое файла `ca.crt` в поле `kubernetes_cert`:

```
kubectl get secret $TOKENNAME -o yaml | grep 'ca.crt:'
```

Теперь соберите свое приложение в конвейере Drone и загрузите его в реестр Docker Hub.

На первом этапе, который называется *build*, мы соберем наше клиентское приложение на Node.js. Для выполнения своих этапов Drone использует образы контейнеров, что делает весь процесс очень гибким. Воспользуемся образом Node.js из Docker Hub:

```
pipeline:
  build:
    image: node
    commands:
      - cd frontend
      - npm i redis --save
```

По завершении сборки вам следует протестировать ее результат, поэтому мы добавим этап *test*, на котором к свежесобранному приложению будет применена команда `npm test`:

```
test:
    image: node
    commands:
      - cd frontend
      - npm i redis --save
      - npm test
```

Вслед за успешной сборкой и тестированием приложения вам нужно выполнить шаг *publish*, чтобы создать для своего приложения образ Docker и загрузить его в Docker Hub.

Внесите следующие изменения в файл `.drone.yml`:

```
repo: <your-registry>/frontend

publish:
    image: plugins/docker
    dockerfile: ./frontend/Dockerfile
    context: ./frontend
    repo: dstrebel/frontend
    tags: [latest, v2]
    secrets: [ docker_username, docker_password ]
```

В результате выполнения этого этапа образ будет загружен в ваш реестр Docker.

Подготовка CD

На этапе развертывания вы загрузите приложение в свой кластер Kubernetes. Используйте манифест развертывания, который находится в папке клиентского приложения (frontend) внутри вашего репозитория:

```
kubectl:
    image: dstrebel/drone-kubectl-helm
    secrets: [ kubernetes_server, kubernetes_cert, kubernetes_token ]
    kubectl: "apply -f ./frontend/deployment.yaml"
```

Как только приложение будет развернуто, в вашем кластере появятся новые pod. Чтобы в этом убедиться, выполните следующую команду:

```
kubectl get pods
```

Вы также можете предусмотреть этап тестирования, чтобы получить состояние `Deployment`. Добавьте следующий раздел в конвейер Drone:

```
test-deployment:
  image: dstrebel/drone-kubectl-helm
  secrets: [ kubernetes_server, kubernetes_cert, kubernetes_token ]
  kubectl: "get deployment frontend"
```

Выполнение плавающего обновления

Поменяем строчку в коде клиентского приложения, чтобы продемонстрировать плавающее обновление. Откройте файл `server.js`, отредактируйте следующий участок и зафиксируйте изменение:

```
console.log('api server is running.');
```

Вы увидите, как выкатываются новые pod и происходит плавающее обновление. В результате будет развернута новая версия вашего приложения.

Простой хаотический эксперимент

В экосистеме Kubernetes существует целый ряд инструментов, с помощью которых можно проводить хаотические эксперименты. В их число входят как эффективные удаленные решения типа CaaS (chaos as a service — «хаос как услуга»), так и простые утилиты для удаления pod в вашей среде. Ниже перечислены инструменты, которые, как показывает наш опыт, дают положительные результаты.

- *Gremlin* — удаленный сервис, предоставляющий продвинутые возможности для проведения хаотических экспериментов.
- *PowerfulSeal* — открытый проект со сложными хаотическими сценариями.
- *Chaos Toolkit* — открытый проект, миссия которого заключается в предоставлении бесплатного, свободного набора инструментов и API для

разнообразных средств хаотического тестирования. Разрабатывается сообществом.

- *KubeMonkey* — открытый инструмент, который позволяет проводить простое тестирование устойчивости pod в кластере.

Подготовим короткий хаотический эксперимент, чтобы проверить устойчивость нашего приложения путем автоматического удаления pod. Воспользуемся для этого Chaos Toolkit:

```
pip install -U chaostoolkit

pip install chaostoolkit-kubernetes

export FRONTEND_URL="http://$(kubectl get svc frontend -o
 jsonpath="{.status.loadBalancer.ingress[*].ip}"):8080/api/"

chaos run experiment.json
```

Рекомендации относительно CI/CD

Не ожидайте, что вам сразу удастся наладить идеальный процесс CI/CD. Для его постепенного улучшения следуйте перечисленным ниже рекомендациям.

- При настройке CI делайте акцент на автоматизации и быстрой сборке. Чем быстрее происходит этот процесс, тем скорее разработчики смогут узнать о том, что их изменения нарушили сборку приложения.
- Сосредоточьтесь на повышении надежности своих тестов. Это позволит разработчикам быстро выявлять проблемы в своем коде. Чем быстрее обратная связь, тем более высокой будет производительность труда разработчиков.
- При выборе инструментов CI/CD убедитесь в том, что они позволяют описывать процесс в виде кода. Так вы сможете управлять версиями этого процесса с помощью кода своего приложения.
- Не забывайте оптимизировать свои образы, сокращая их размер и уменьшая поверхность атаки во время их выполнения в промышленных условиях. Docker поддерживает многоэтапную сборку, в процессе которой можно удалять пакеты, не участвующие в выполнении приложения. Например, утилита Maven позволит собрать код, но для работы полученного образа она не нужна.
- Избегайте использования тега `latest`. Применяйте *теги*, которые ссылаются на идентификатор сборки или коммит кода в Git.

- Если вы только знакомитесь с непрерывным развертыванием, то на первых порах вам лучше остановиться на плавающих обновлениях. Они просты в использовании и позволят освоиться с процессом CD. Приобретя определенный опыт, вы сможете попробовать стратегии с сине-зелеными и канареечными развертываниями.
- При использовании CD всегда проверяйте, как обновление вашего приложения сказывается на клиентских соединениях и схеме базы данных.
- Тестирование в промышленных условиях позволит вам сделать свое приложение более надежным и обеспечит наличие качественного мониторинга. Начинайте проведение своих экспериментов с небольшого масштаба и ограничивайте потенциальные последствия.

Резюме

В данной главе мы обсудили этапы процесса CI/CD, призванного обеспечить надежную доставку программного обеспечения. CI/CD помогает снизить риск и повысить скорость развертывания приложений в Kubernetes. Мы также рассмотрели разные стратегии обновления, которые можно применять для доставки кода.

ГЛАВА 6

Версии, релизы и выкатывание обновлений

Один из главных недостатков традиционных монолитных приложений состоит в том, что со временем они становятся слишком большими и громоздкими для обновления, ведения версий и внесения изменений в сроки, соответствующие бизнес-требованиям. Многие считают, что это был один из определяющих факторов, которые привели к внедрению более динамичных подходов к разработке и распространению микросервисных архитектур. Возможность быстро доставлять новый код, решать свежие проблемы и исправлять скрытые ошибки еще до возникновения серьезных последствий, а также заявленные обновления с нулевым временем простоя — это все то, к чему стремятся команды разработчиков в этом постоянно меняющемся мире интернет-экономики. На самом деле упомянутые проблемы можно решить в системе любого типа, применяя подходящие процессы и процедуры, но обычно это требует куда более существенных технологических и человеческих ресурсов.

Внедрение контейнеров в качестве среды выполнения программного кода обеспечило изоляцию и компонуемость, которые позволили проектировать довольно динамичные и надежные системы. Но при этом поддержка этих характеристик в крупных масштабах по-прежнему нуждалась в высоком уровне человеческого вмешательства. По мере роста системы становились все более хрупкими, требуя построения сложных процессов для автоматизации механизмов создания новых релизов, выполнения обновлений и обнаружения сбоев. Средства оркестрации сервисов, такие как Apache Mesos, HashiCorp Nomad и даже специализированные контейнерные инструменты наподобие Kubernetes и Docker Swarm, инкапсулировали эти механизмы в виде более тривиальных компонентов для своих сред выполнения. Теперь системные инженеры могут решать более сложные проблемы, имея в своем распоряжении средства для управления версиями, создания релизов и развертывания приложений в системе.

Ведение версий

Текущий раздел не является введением в управление версиями ПО и историю данного процесса. На эту тему написаны многочисленные статьи и учебники по информатике. Главное — выбрать какой-то один подход и четко ему следовать. Большинство компаний и разработчиков, занимающихся созданием программного обеспечения, согласны с тем, что *семантический* принцип ведения версий наиболее полезен; особенно это касается микросервисной архитектуры, в которой команды, отвечающие за написание разных микросервисов, должны обеспечивать их совместимость на уровне API.

Если вы незнакомы с семантическим ведением версий, то основная идея заключается в том, что номер версии состоит из трех частей — *мажорной*, *минорной* и *патча*, разделенных, как правило, *точками*: *1(мажорная).2(минорная).3(патч)*. Патч обозначает инкрементальный релиз с исправлением ошибки или очень незначительным изменением, которое не влияет на API. Минорная версия обозначает обновления, которые могут приводить к изменениям в API, но притом являются обратно совместимыми с предыдущим релизом. Это имеет большое значение для разработчиков, взаимодействующих с другими микросервисами, в написании которых не участвуют. Если мы работаем с микросервисом версии 1.4.7, недавно обновившимся до версии 1.4.8, то можем быть уверены в том, что нам нужно менять свой код только при наличии желания использовать новые возможности API. Мажорная версия привносит в код несовместимые изменения. В большинстве случаев разные мажорные версии одного и того же кода не могут работать с одним и тем же API. Данная система имеет множество нюансов. Например, жизненный цикл разработки разбивается на четыре этапа: скажем, 1.4.7.0 — это альфа-версия, а 1.4.7.3 — готовый релиз. Самое главное, что во всей системе соблюдается один и тот же подход.

Релизы

По правде говоря, платформа Kubernetes сама по себе не поддерживает концепцию релизов и не предоставляет подходящих контроллеров. Информация о релизе обычно указывается в поле `Deployment metadata.labels` и/или в поле pod `pod.spec.template.metadata.label`. То, в каких ситуациях используется некое значение, очень важный вопрос и может иметь разные последствия в зависимости от того, каким образом процесс CD выкатывает обновления. Одной из основных концепций, которые диспетчер Helm

привнес в экосистему Kubernetes, было понятие релиза, позволившее различать экземпляры чартов, запущенных в кластере. Тот же подход можно реализовать и самостоятельно, однако Helm отслеживает выпуски и их историю и используется многими инструментами непрерывного развертывания в качестве стандартного сервиса для управления релизами. Опять же главное здесь — быть последовательными в работе с версиями и в том, как этот процесс сказывается на состоянии кластера.

В вашей организации могут действовать правила относительно названий релизов. Для этого часто используются метки, например `stable` или `canary`, которые помогают придать системе некую управляемость при добавлении средств гибкой маршрутизации, таких как механизмы межсервисного взаимодействия. Большие организации, выкатывающие многочисленные изменения для разных аудиторий, зачастую внедряют кольцевую архитектуру с обозначениями наподобие `ring-0`, `ring-1` и т. д.

Чтобы вы лучше понимали, о чем речь, следует немного отвлечься и поговорить об особенностях использования меток в декларативной модели Kubernetes. Метки как таковые представляют собой произвольные пары «ключ — значение», которые подчиняются синтаксическим правилам API. Главное здесь не само содержимое, а то, как каждый контроллер работает с метками, обрабатывает их изменение и сопоставляет их с селекторами. Такие объекты, как `Job`, `Deployment`, `ReplicaSet` и `DaemonSet`, поддерживают поиск pod по их меткам путем прямого сопоставления или с помощью выражений на основе множеств. Необходимо понимать: селекторы, основанные на метках, нельзя изменить после их создания. Это означает, что при добавлении нового селектора, который соответствует меткам каких-либо pod, существующий объект `ReplicaSet` не обновляется, а вдобавок к нему создается новый. Данный факт имеет огромное значение в процессе развертывания обновлений, который мы обсудим дальше.

Развертывание обновлений

До того как в Kubernetes появился контроллер `Deployment`, единственным механизмом для управления процессом развертывания приложений была консольная команда `kubectl rolling-update`, которой нужно было передавать название обновляемого контроллера `replicaController`. Этот подход было сложно интегрировать в декларативную модель CD, поскольку он не являлся частью состояния исходного манифеста. Администраторам приходилось

тщательно следить за корректным обновлением манифестов и ведением версий, чтобы случайно не откатить систему назад и иметь возможность архивировать ее, если она больше не была нужна. Контроллер `Deployment` позволил автоматизировать данный процесс обновления, используя подходящую стратегию и позволяя системе считывать новое декларативное состояние, основанное на изменениях в манифесте `Deployment spec.template`. Новички в Kubernetes часто неправильно интерпретируют этот последний факт, из-за чего получаются досадные результаты, когда новички меняют метку в поле развертывания `metadata`, заново применяют манифест и видят, что обновление не было инициировано. Контроллер `Deployment` способен обнаружить изменения, внесенные в спецификацию, и принять меры по обновлению приложения в соответствии со стратегией, которая указана в данной спецификации. Kubernetes поддерживает две стратегии: `rollingUpdate` (используется по умолчанию) и `recreate`.

Если указать плавающее обновление (`rollingUpdate`), то в ходе развертывания будет создан новый объект ReplicaSet с нужным количеством реплик, а старый — сведен к нулю с учетом значений `maxUnavailble` и `maxSurge`. Эти два значения, в сущности, не дают Kubernetes удалить старые pod, пока не запустится достаточное количество новых реплик; при этом новые реплики начнут создаваться только после удаления определенного количества старых pod. Данный процесс имеет одну замечательную особенность: контроллер `Deployment` хранит историю обновлений, поэтому вы можете откатываться на предыдущие версии, используя командную строку.

Стратегия `recreate` подходит для определенных рабочих заданий, которые способны справляться с серьезными перебоями в работе pod в `ReplicaSet`, не допуская притом существенного ухудшения в качестве обслуживания. В этой модели контроллер `Deployment` создает новый объект `ReplicaSet` с новой конфигурацией и удаляет старый еще до того, как делает доступными новые pod. С такого рода ситуациями, к примеру, могут справляться сервисы, запросы к которым проходят через системы очередей: запрос попадет в очередь и дождется, когда новый pod станет доступным, после чего его обработка продолжится.

Полноценный пример

Ведение версий, создание релизов и управление процессом развертывания влияют сразу на несколько ключевых аспектов отдельно взятого объекта

Deployment. Рассмотрим пример манифеста и затем пройдемся по интересующим нас участкам в контексте рекомендованных подходов:

```
# развертывание веб-приложения
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gb-web-deploy
  labels:
    app: guest-book
    appver: 1.6.9
    environment: production
    release: guest-book-stable
    release number: 34e57f01
spec:
  strategy:
    type: rollingUpdate
    rollingUpdate:
      maxUnavailbale: 3
      maxSurge: 2
  selector:
    matchLabels:
      app: gb-web
      ver: 1.5.8
    matchExpressions:
      - {key: environment, operator: In, values: [production]}
  template:
    metadata:
      labels:
        app: gb-web
        ver: 1.5.8
        environment: production
    spec:
      containers:
      - name: gb-web-cont
        image: evillgenius/gb-web:v1.5.5
        env:
        - name: GB_DB_HOST
            value: gb-mysql
        - name: GB_DB_PASSWORD
            valueFrom:
                secretKeyRef:
                  name: mysql-pass
                  key: password
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 80
---
```

```
# развертывание БД
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gb-mysql
  labels:
    app: guest-book
    appver: 1.6.9
    environment: production
    release: guest-book-stable
    release number: 34e57f01
spec:
  selector:
    matchLabels:
      app: gb-db
      tier: backend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: gb-db
        tier: backend
        ver: 1.5.9
        environment: production
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
---
# задание для резервного копирования БД
apiVersion: batch/v1
kind: Job
metadata:
  name: db-backup
```

```
  labels:
    app: guest-book
    appver: 1.6.9
    environment: production
    release: guest-book-stable
    release number: 34e57f01
  annotations:
    "helm.sh/hook": pre-upgrade
    "helm.sh/hook": pre-delete
    "helm.sh/hook": pre-rollback
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  template:
    metadata:
      labels:
        app: gb-db-backup
        tier: backend
        ver: 1.6.1
        environment: production
    spec:
      containers:
      - name: mysqldump
        image: evillgenius/mysqldump:v1
        env:
        - name: DB_NAME
          value: gbdb1
        - name: GB_DB_HOST
          value: gb-mysql
        - name: GB_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        volumeMounts:
          - mountPath: /mysqldump
            name: mysqldump
      volumes:
      - name: mysqldump
        hostPath:
          path: /home/bck/mysqldump
    restartPolicy: Never
backoffLimit: 3
```

На первый взгляд этот пример может показаться немного странным. Из-за чего у объекта `Deployment` и образа контейнера разные теги `version`? Что произойдет, если поменяется только один из них, а другой останется прежним? Какую роль здесь играет поле `release` и как повлияет на систему его изменение? Если поменяется определенная метка, то когда начнется обновление данного объекта `Deployment`? Чтобы ответить на эти вопросы,

рассмотрим некоторые рекомендации по ведению версий, созданию релизов и развертыванию обновлений.

Рекомендации по ведению версий, созданию релизов и развертыванию обновлений

Эффективный процесс CI/CD и поддержка развертывания с сокращенным или нулевым временем простоя требуют согласованности в ведении версий и управлении релизами. Рекомендации, приведенные ниже, помогут определить согласованные параметры, с помощью которых команды DevOps смогут выполнять беспроблемные и незаметные развертывания.

- Семантические версии должны относиться ко всему приложению и отличаться от версий контейнеров и pod, из которых оно состоит. Это позволяет сделать жизненные циклы контейнеров и самого приложения независимыми. Поначалу такой подход может показаться довольно запутанным, но, установив принципиальные правила относительно того, как изменения одной версии влекут за собой обновление другой, вы сможете легко разобраться в этом. В предыдущем примере контейнер сам по себе имел версию v1.5.5, в то время как в спецификации pod была указана версия v1.5.8. Это значит, что спецификация могла измениться (например, в нее могли добавить объекты `ConfigMap`, новые секретные данные или обновленные значения реплик), но этот конкретный контейнер сохранил старую версию. Само приложение guest-book и все его сервисы имели версию 1.6.9; это может означать, что на протяжении какого-то времени инженеры вносили изменения, выходящие за рамки этого отдельного сервиса (например, они могли касаться других сервисов, из которых состоит приложение).
- Используйте метки `release` и `release version/number` в метаданных своего `Deployment`, чтобы отслеживать релизы в ходе CI/CD. Название и номер релиза должны совпадать с тем релизом, который записывает средство CI/CD. Это позволяет просматривать все этапы процесса CI/CD в кластере и упрощает идентификацию `Deployment`. В предыдущем примере номер релиза основан непосредственно на идентификаторе процесса CD, создающего манифест.
- Если перед развертыванием в Kubernetes сервисы упаковываются с помощью Helm, то уделите особое внимание объединению в общий чарт тех из них, которые нужно откатывать или обновлять вместе. Helm позволяет

с легкостью откатывать все компоненты приложения, возвращая его в состояние, наблюдавшееся до обновления. Поскольку Helm обрабатывает шаблоны и все свои директивы до того, как передавать конфигурацию в формате YAML, благодаря хукам жизненного цикла обеспечивается правильный порядок выполнения операций в этих шаблонах. Применяя подходящие хуки жизненного цикла Helm, инженеры могут сделать так, чтобы обновление и откат происходили корректно. В предыдущем примере спецификация `Job` использовала хуки для того, чтобы шаблон выполнял резервное копирование базы данных до отката, обновления или удаления релиза Helm. Благодаря этому объект `Job` удаляется только после успешного завершения задания; к сожалению, это значит, вам придется удалять его вручную, поскольку контроллер TTL в Kubernetes все еще находится на стадии альфа-тестирования.

- Согласуйте ту схему релизов, которая соответствует темпу обновлений в вашей организации. Для большинства ситуаций подойдут такие состояния, как `stable`, `canary` и `alpha`.

Резюме

Kubernetes позволяет крупным и мелким компаниям внедрять более сложные и гибкие процессы разработки. Возможность автоматизировать трудоемкие процессы, которые раньше требовали большого количества человеческих и технических ресурсов, стала общедоступной. Теперь даже стартапы могут относительно легко пользоваться преимуществами этого облачного подхода. Декларативная природа Kubernetes по-настоящему проявляет себя при подборе подходящей модели использования меток и применении возможностей стандартных контроллеров этой платформы. Если правильно описать этапы разработки и эксплуатации с помощью декларативных свойств приложений, развертываемых в Kubernetes, то это поможет вашей организации управлять такими сложными процессами, как обновление, развертывание и откат изменений за счет применения инструментария и автоматизации.

ГЛАВА 7

Глобальное распределение приложений и промежуточное тестирование

До сих пор в этой книге приводились различные рекомендации по сборке, разработке и развертыванию приложений, однако доставка и администрирование сервисов в глобальном масштабе создают совершенно новые вызовы.

Глобальное развертывание приложения может быть обусловлено множеством разных причин. Первая и самая очевидная заключается в масштабе. Ваше приложение может быть настолько успешным или важным, что его просто необходимо развернуть по всему миру, чтобы его производительности было достаточно для обслуживания всех его пользователей. Примеры таких приложений — глобальный API шлюза для публичного облачного провайдера, крупномасштабный IoT-продукт с глобальным охватом, крайне успешная социальная сеть и т. д.

Систем с подобным масштабом не так уж много, но к ним следует прибавить приложения, которые используют глобальное присутствие для уменьшения латентности. Даже контейнеры и Kubernetes не могут преодолеть скорость света, поэтому в целях минимизации времени ответа системы иногда приходится распределять так, чтобы они находились поближе к пользователям.

И наконец, еще более распространенная причина для глобального распределения — географические особенности. Они могут быть связаны с пропускной способностью (как в случае с удаленными сенсорными платформами), конфиденциальностью данных (региональные ограничения) или тем фактом, что для достижения успеха некоторые приложения необходимо развертывать в определенном регионе.

Во всех упомянутых случаях ваша система больше не ограничена горсткой промышленных кластеров. Вместо этого она распределяется между десятками

или сотнями разных географических областей, управление которыми, равно как и требования к предоставлению услуг на глобальном уровне, представляют собой серьезный вызов. В данной главе рассматриваются подходы и рекомендации, которые помогут справиться с этим.

Распределение вашего образа

Прежде чем всерьез задумываться о глобальном распределении своего приложения, вам нужно сделать его образ доступным в кластерах, размещенных по всему миру. Первым делом следует убедиться в том, что в вашем реестре образов включена автоматическая георепликация. Многие реестры, предоставляемые облачными провайдерами, сами занимаются глобальным распределением вашего образа; если его попытается загрузить какой-то кластер, то он будет взят из ближайшего хранилища. Многие облака оставляют выбор включения репликации образов за пользователем; например, вам может быть известно о регионах, в которых вы не собираетесь присутствовать. Один из таких реестров — Azure Container Registry (https://azure.microsoft.com/ru-ru/services/container-registry/), но у него есть конкуренты с аналогичными возможностями. Если вы используете реестр облачного провайдера с поддержкой георепликации, то распределение вашего образа по всему миру не вызовет никаких трудностей. Вам достаточно загрузить образ в реестр, выбрать регионы для географического распределения, а об остальном позаботится ваш провайдер.

Если вы не используете облачный реестр или ваш провайдер не поддерживает автоматическую георепликацию образов, то вам придется решить эту задачу самостоятельно. Для этого можно выбрать реестр, размещенный в определенном месте. Но такой подход чреват несколькими проблемами. Латентность загрузки образа зачастую определяет то, насколько быстро контейнер может быть запущен в кластере. От этого, в свою очередь, зависит время вашей реакции на аппаратные сбои, учитывая, что в таких случаях обычно необходимо переносить образы контейнеров на новый сервер.

Еще одним недостатком использования одного реестра является тот факт, что реестр может стать единой точкой отказа. Если он размещен в одном регионе или вычислительном центре, то может оказаться недоступным из-за серьезного локального происшествия. Без реестра перестанет работать ваш процесс CI/CD и вы потеряете возможность развертывать новый код. Это, несомненно, существенно повлияет на производительность труда разработчиков и на функционирование приложения. Кроме того, единый

реестр может быть очень затратным, поскольку при запуске каждого нового контейнера будет расходоваться большой объем трафика; и хотя образы контейнеров довольные компактные, их совокупный размер со временем только увеличивается. Однако, несмотря на все эти недостатки, использование одного реестра может оказаться адекватным решением для небольших приложений, которые работают всего в нескольких глобальных регионах. В любом случае его намного проще сконфигурировать, чем полномасштабную репликацию образов.

Если вам нужно распределить свой образ, но вы не можете использовать георепликацию от облачных провайдеров, то у вас остается лишь один вариант: реализовать собственное решение. Это можно сделать двумя путями. Первый состоит в использовании географических названий для каждого отдельного реестра (например, `us.my-registry.io`, `eu.my-registry.io` и т. д.). Преимущество этого подхода — простота в настройке и администрировании. Все реестры остаются полностью независимыми, и вы можете загружать свои образы в каждый из них на последнем этапе процесса CI/CD. Недостаток подхода связан с тем, что для загрузки образов из ближайшей географической области конфигурация каждого кластера должна немного отличаться. Но, учитывая, что у конфигурации вашего приложения, скорее всего, все равно будут географические расхождения, эта проблема уже, вероятно, присутствует в вашей среде, и ее относительно легко нивелировать.

Параметризация развертываний

После репликации вашего образа вам необходимо параметризировать свои развертывания для разных глобальных областей. Когда развертывание происходит в ряде разных регионов, в конфигурации приложения всегда возникают какие-то расхождения. Например, при отсутствии реестра с георепликацией вам, возможно, придется менять название образа в зависимости от региона. Но, даже если георепликация доступна, вполне вероятно, что в разных географических областях ваше приложение будет испытывать разные нагрузки; следовательно, вам могут понадобиться разное количество реплик и другие параметры. Преодоление этих трудностей без затраты чрезмерных усилий — ключ к успешному администрированию глобальных приложений.

Первое, о чем необходимо позаботиться, — это подготовка разных конфигураций на диске. Обычно для этого каждому глобальному региону выделяют отдельный каталог. Конечно, вы можете просто скопировать в разные

каталоги одну и ту же конфигурацию, но это неминуемо приведет к тому, что вы однажды забудете обновить некоторые регионы, чем вызовете расхождения в их конфигурационных файлах. Вместо этого лучше всего применять подход на основе шаблонов: большая часть настроек размещается в общем файле, из которого с помощью параметров генерируется конфигурация для отдельных регионов. Для такого рода шаблонизации, как правило, используют Helm (helm.sh; ищите подробности в главе 2).

Глобальное распределение трафика

Распределив свое приложение по всему миру, вы должны подумать о том, как направлять к нему трафик. В целом, чтобы доступ к вашему сервису имел низкую латентность, необходимо учитывать географическую удаленность. Но в то же время следует позаботиться об отказоустойчивости, чтобы в случае отказа какой-то одной области или другого рода проблем вы могли перенаправить ваш трафик. Правильное распределение трафика между разными регионами — ключ к созданию производительной и надежной системы.

Предположим, что вы хотите предоставлять свои услуги через единый домен, такой как myapp.myco.com. В самом начале вам нужно определиться с тем, хотите ли вы использовать протокол DNS (Domain Name System — система доменных имен) для распределения трафика между своими региональными конечными точками. Если вы будете применять DNS для балансировки нагрузки, то пользовательские DNS-запросы к myapp.myco.com будут возвращать разные IP-адреса в зависимости от местоположения пользователей, обращающихся к вашему сервису, и от текущей доступности ваших кластеров.

Надежное развертывание программного обеспечения в глобальном масштабе

Шаблонизировав свое приложение так, чтобы оно имело подходящую конфигурацию в каждом регионе, вы должны решить еще одну проблему: как развертывать эти конфигурационные файлы по всему миру? Самое очевидное решение состоит в том, чтобы глобально доставлять все приложение целиком; это будет способствовать его эффективному и быстрому развитию. Но такой подход, несмотря на свою гибкость, может легко привести к глобальному

сбою. Вместо этого для большинства промышленных приложений лучше применять промежуточное выкатывание. В сочетании с такими функциями, как глобальная балансировка нагрузки, это обеспечит высокую доступность даже в случае серьезных неполадок.

Подходя к проблеме глобального развертывания, вы должны уделять основное внимание и скорости доставки ПО, и быстрому обнаружению проблем (желательно до того, как они затронут других пользователей). Предположим, что к моменту проведения глобального развертывания ваше приложение уже прошло базовое функциональное и нагрузочное тестирование. Прежде чем сертифицировать образ (или образы) для промышленного использования по всему миру, вы должны убедиться в том, что он прошел все необходимые проверки и приложение функционирует корректно. Необходимо отметить: это вовсе *не гарантирует* корректную работу вашего кода. Тестирование помогает выявить много проблем, но некоторые из них проявляются лишь во время обслуживания промышленного трафика. Это связано с тем, что реальные нагрузки зачастую сложно имитировать идеально. Например, вы можете проверить ввод только для англоязычных пользователей, тогда как в реальности приложение будет получать данные на разных языках. Или же проверяемый вами ввод не охватывает все возможные случаи, с которыми сталкивается промышленное приложение. Каждый раз, когда в вашей промышленной среде возникает проблема, не проявлявшаяся во время тестирования, это верный признак того, что вам необходимо расширить и разнообразить свои тесты. Тем не менее факт остается фактом: многие проблемы обнаруживаются лишь в ходе промышленного развертывания.

Учитывая вышесказанное, в каждом регионе, в котором вы проводите развертывание, могут возникать свои проблемы. И поскольку речь идет о промышленных средах, это чревато сбоями, на которые вам придется реагировать. Сочетание данных факторов обуславливает то, как вы должны подходить к региональным развертываниям.

Проверка перед развертыванием

Прежде чем браться за глобальное развертывание конкретной версии программного обеспечения, ее необходимо проверить в какой-либо синтетической тестовой среде. Если вы корректно настроили свой процесс CD, то весь код перед релизом той или иной сборки проходит модульное и, возможно,

ограниченное интеграционное тестирование. Но даже в этом случае, прежде чем выкатывать данный релиз, следует рассмотреть необходимость проведения двух других видов теста. Прежде всего это полноценное интеграционное тестирование; вы должны выполнить полномасштабное развертывание приложения вместе со всем его стеком, но без реального трафика. В стек обычно входят данные, скопированные из промышленной среды или сгенерированные в соответствии с реальными размерами и масштабами. Если в реальных условиях данные вашего приложения занимают 500 Гбайт, то очень важно, чтобы в ходе промежуточного тестирования вы использовали набор данных примерно того же размера (а лучше пусть это будут идентичные данные).

Это, по большому счету, самая сложная часть подготовки среды для полноценного интеграционного тестирования. Промышленные данные зачастую существуют только в промышленной среде, а сгенерировать синтетический набор данных того же размера и масштаба довольно сложно. Учитывая это, создание реалистичного набора данных для интеграционного тестирования — отличный пример задачи, которую лучше решить на ранних этапах разработки приложения. В самом начале ваша синтетическая копия будет относительно небольшой и станет увеличиваться теми же темпами, что и ваши промышленные данные. Работать с ней будет намного проще, чем пытаться дублировать промышленные данные в крупных масштабах.

К сожалению, многие люди не понимают, что данные нужно копировать, пока это легко сделать — до того, как система разрастется. Но если уже поздно, то вы можете попробовать развернуть поверх своего промышленного хранилища слой для дублирования чтения/записи. Конечно, вряд ли вам захочется, чтобы ваши интеграционные тесты выполняли запись в промышленной среде, но при этом вы можете разместить перед своим промышленным хранилищем прокси-сервер, который будет читать его содержимое, но выполнять запись во вспомогательную таблицу; последняя также вполне пригодна для всех последующих операций чтения.

Независимо от того, каким образом вы настроите свою среду для интеграционных тестов, цель остается прежней: убедиться в том, что приложение реагирует на тестовый ввод и взаимодействие ожидаемым образом. Определить и выполнить эти тесты можно множеством разных способов: от ручного выполнения набора тестов (чревато ошибками, поэтому не рекомендуется) до имитации взаимодействия пользователя и браузера, включая щелчки кнопкой мыши и т. д. Компромиссное решение заключается в том, чтобы тестировать API-вызовы в формате REST, не уделяя особого внимания

веб-интерфейсу, основанному на них. Независимо от выбранного подхода вы должны преследовать одну цель: создание набора автоматизированных тестов, которые проверяют корректность поведения вашего приложения в ответ на полноценный диапазон реальных входящих данных. В простых проектах эту проверку можно проводить непосредственно перед слиянием веток репозитория, но для большинства реальных крупномасштабных приложений требуется полноценная среда интеграционного тестирования.

Интеграционные тесты могут подтвердить корректность работы вашего приложения, но в дополнение к ним следует также проводить нагрузочное тестирование. Одно дело — продемонстрировать корректную работу вашего кода, и совсем другое — доказать, что он способен выдержать реальную нагрузку. В любой крупномасштабной системе существенное ухудшение производительности (например, повышение латентности на 20 %) заметно влияет на качество взаимодействия с приложением, что может не только разочаровать пользователей, но и привести к серьезным сбоям. В связи с этим необходимо позаботиться о том, чтобы подобное ухудшение производительности не возникло в промышленной среде.

Как и в случае с интеграционными тестами, выработка правильного подхода к нагрузочному тестированию может оказаться непростой задачей, ведь для этого необходимо генерировать синтетическую нагрузку, аналогичную промышленной, но так, чтобы ее можно было воспроизвести. Один из самых простых способов добиться этого заключается в воспроизведении журнальных записей с реальными запросами, взятых из промышленной среды. Это может быть отличным решением для нагрузочного тестирования, характеристики которого соответствуют реальным нагрузкам на развернутое приложение. Однако воспроизведение не всегда дает надежные результаты. Например, если ваши журнальные записи устарели, а ваше приложение или набор данных изменились, то может случиться так, что воспроизводимый и свежий трафик дадут разную нагрузку. Кроме того, при наличии реальных зависимостей, макеты для которых вы не предусмотрели, может возникнуть ситуация, когда старый трафик, отправляемый этим зависимостям, уже недействителен (например, данные могут больше не существовать).

С учетом всех перечисленных трудностей многие системы, даже критически важные, долгое время разрабатываются без нагрузочных тестов. Как и в случае с моделированием промышленных данных, это явный пример процесса, сопровождение которого можно облегчить за счет раннего внедрения. Предусмотреть нагрузочные тесты, когда у приложения все еще

мало зависимостей, и постепенно развивать их по ходу внедрения системы будет намного проще, чем пытаться интегрировать нагрузочное тестирование в существующий крупномасштабный проект.

Вслед за созданием нагрузочного теста следует определиться с тем, какие метрики нужно отслеживать при его выполнении. Очевидно, что в их число должны входить частота и латентность запросов, поскольку эти показатели явно влияют на взаимодействие с пользователями.

При измерении латентности необходимо понимать, что это не какая-то одна величина и вы должны вычислить как среднее значение, так и процентили крайних случаев (например, 90-й и 99-й), поскольку они представляют худшие сценарии взаимодействия с вашим приложением. Иногда средние значения не позволяют выявить проблемы с очень длинными задержками, но если 10 % ваших пользователей испытывают высокую латентность, то это может заметно сказаться на успехе вашего продукта.

Кроме того, во время нагрузочного тестирования следует обращать внимание на ресурсы, потребляемые приложением (процесс, память, сеть, диск). Эти метрики не имеют прямого отношения к взаимодействию с пользователями, однако резкие скачки в потреблении ресурсов должны быть выявлены и проанализированы до развертывания в промышленной среде. Если ваше приложение внезапно удваивает расход памяти, то данный факт стоит исследовать, даже если нагрузочные тесты были пройдены, поскольку рано или поздно такая чрезмерная расточительность повлияет на качество работы и доступность вашей системы. В зависимости от обстоятельств развертывание релиза можно продолжить, но в то же время вам необходимо разобраться, с чем связаны изменения в потреблении ресурсов.

Канареечный регион

Если в ходе тестирования ваше приложение вело себя корректно, следующим шагом должно быть развертывание в *канареечном регионе*. Он получает трафик от настоящих пользователей и команд, желающих проверить ваш релиз. Это могут быть как внутренние команды, которые вы обслуживаете, так и внешние клиенты, пользующиеся вашим сервисом. Такие регионы нужны для того, чтобы заранее предупредить пользователей о выкатываемых изменениях, которые могут повлиять на их работу. Какими бы эффективными ни были ваши интеграционные и нагрузочные тесты, все-

гда существует вероятность того, что вы упустили из виду некую ошибку, чреватую созданием больших проблем для тех или иных пользователей или клиентов. Подобные ошибки лучше отлавливать в среде, в которой все, кто использует ваш сервис или развертывает код, взаимодействующий с ним, осознает повышенный риск возникновения сбоев. Для этого и нужен канареечный регион.

К канареечным развертываниям следует относиться как к промышленным; это касается мониторинга, масштаба, функций и т. д. Но, поскольку это первая остановка на пути к релизу, здесь чаще всего сталкиваются с проблемами, препятствующими развертыванию. Такое положение дел нормально; на самом деле в этом весь смысл. Ваши клиенты добровольно соглашаются использовать канареечный регион для задач, не несущих больших рисков (таких как разработка или обслуживание внутренних пользователей), чтобы заранее узнавать о любых несовместимостях в будущих релизах.

Поскольку цель канареечных развертываний — получение обратной связи на ранних этапах, будет лучше, если релиз проведет в канареечном регионе несколько дней. Это позволит поработать с ним более широкому кругу клиентов, прежде чем он будет развернут в других регионах. Такие сроки объясняются тем, что отдельные ошибки являются вероятностными (например, могут возникать в 1 % запросов) или выдают себя только в крайних случаях, для возникновения которых может потребоваться какое-то время. Проблема может оказаться настолько незначительной, что будет проигнорирована системой автоматических уведомлений, но при этом чревата некорректным поведением бизнес-логики, заметить которое можно лишь при взаимодействии с клиентами.

Разные типы регионов

Прежде чем приступать к глобальному развертыванию своего ПО, необходимо подумать об отличительных характеристиках ваших регионов. В ходе доставки в промышленную среду ваш код должен пройти через интеграционное тестирование и начальную проверку в канареечном регионе. Это значит, что впоследствии любые найденные вами проблемы не проявили себя ни в одном из этих окружений. Подумайте, чем отличаются ваши регионы. Возможно, одни из них принимают больше трафика, чем другие. Может быть, к некоторым из них обращаются особым образом. Например,

в развивающихся странах запросы с большей долей вероятности отправляются из мобильных браузеров. В связи с этим регион, находящийся рядом с большим количеством развивающихся стран, будет иметь намного больше мобильного трафика, чем тестовые или канареечные регионы.

Еще один пример — язык ввода. В неанглоязычных регионах пользователи могут отправлять больше текста в формате Unicode, вследствие чего возможно возникновение ошибок в обработке строк или символов. Если в основе вашего сервиса лежит API, то некоторые части этого интерфейса могут пользоваться большей популярностью в определенных регионах. Все это — примеры отличий, которые могут проявляться за пределами вашей пробной среды. Каждый такой случай — потенциальный источник происшествий в промышленной системе. Создайте таблицу с разными характеристиками, которые, как вам кажется, заслуживают внимания. Это поможет в планировании вашего глобального развертывания.

Подготовка к глобальному развертыванию

Определив характеристики своих регионов, вы должны составить план развертывания в каждом из них. Очевидно, что вам следует позаботиться о минимизации последствий потенциальных перебоев в работе промышленной среды, поэтому начать стоит с региона, который больше всего похож на канареечный и имеет относительно немного пользователей. Вероятность возникновения проблем в таком регионе крайне низкая, но если они все же возникнут, то последствия будут не слишком серьезными (благодаря меньшим объемам трафика).

После успешного развертывания в первом промышленном регионе вы должны решить, сколько нужно ждать, прежде чем переходить к следующему региону. Ожидание необходимо не для искусственной задержки релиза, а скорее чтобы увидеть первые результаты. Обычно под этим понимают период между завершением развертывания и моментом, когда ваша система мониторинга обнаруживает признаки какой-либо проблемы. Конечно, проблема, если она существует, будет присутствовать в вашей инфраструктуре сразу же после развертывания ПО. Однако, несмотря на это, она может проявиться только спустя какое-то время. Например, ваша система мониторинга или ваши пользователи заметят утечку памяти лишь по прошествии целого часа или больше. Это распределение вероятностей, которое устанавливает, сколько времени должно пройти, прежде чем у вас появится хороший шанс на развертывание корректно работающего релиза. Опыт демонстрирует:

данный показатель обычно вдвое дольше времени, которое уходит на то, чтобы проблема себя проявила.

Если на протяжении последних шести месяцев каждый сбой возникал в среднем через час, то задержка между региональными развертываниями в размере 2 ч должна дать высокую вероятность успешного релиза. Этот показатель можно оценить еще точнее, сформировав более насыщенную (но содержательную) статистику, основанную на истории развертывания вашего приложения.

После успешного развертывания в канареечном регионе с ограниченным трафиком можно переходить к другому канареечному региону с большим количеством пользователей. Здесь будут примерно такие же входящие данные, но в больших объемах. Поскольку единственное отличие от предыдущей процедуры заключается в повышенной нагрузке, все, что вы здесь тестируете, — это способность приложения масштабироваться. Если данное развертывание пройдет без проблем, то вы сможете быть уверены в качестве своего релиза.

Вы должны и дальше следовать тому же принципу для других потенциальных различий в трафике. Например, следующим шагом может быть развертывание в малонагруженном регионе в Азии или Европе. На данном этапе вам, вероятно, захочется ускорить процесс, но помните: каждый раз, когда вы имеете дело с существенными изменениями во входящих данных или нагрузке, начинать следует с единственного региона. Основательно проверив все потенциальные изменения в промышленном регионе своего приложения, вы можете распараллелить процесс выкатывания релиза и быть уверены в том, что он завершится успешно.

Когда что-то идет не так

Мы рассмотрели различные аспекты глобального развертывания программной системы и методы структурирования данного процесса, которые позволяют минимизировать риск возникновения проблем. Но как быть, если что-то вдруг пойдет не так? Все работники аварийно-спасательных служб знают: из-за чрезмерного напряжения в кризисных ситуациях человеку сложно вспомнить даже самые простые инструкции. Прибавьте к этому давление со стороны всех сотрудников компании, начиная с генерального директора, которые ожидают от вас рапорта об успешном устранении проблемы. Очевидно, в подобной ситуации очень легко допустить ошибку.

К тому же в таких условиях даже мелкое упущение (например, пропуск одного из этапов процедуры восстановления) может на порядок ухудшить проблему.

С учетом всего перечисленного очень важно, чтобы ваша реакция на неполадки с развертыванием была быстрой, спокойной и корректной. Для выполнения всего необходимого и в правильном порядке желательно заранее подготовить четкие пошаговые инструкции с описанием результатов, которые следует ожидать на каждом шаге. Пропишите каждый этап, пусть он и кажется очевидным. Сгоряча очень легко забыть и случайно пропустить даже самые простые и тривиальные вещи.

Чтобы не теряться в стрессовых ситуациях, работники аварийно-спасательных служб отрабатывают все процедуры в нормальных условиях. То же самое применимо и к тем действиям, которые необходимо предпринять в случае проблемного развертывания. Вначале нужно определить все шаги, которые требуется выполнить в ответ на возникновение проблемы и для отката на предыдущую версию. В идеале первым делом следует «остановить кровотечение»: перенаправить пользовательский трафик из проблемного региона (-ов) туда, где развертывание еще не проходило и ваша система попрежнему работает корректно. Данную процедуру необходимо отрабатывать в первую очередь. Удается ли вам успешно перенаправить трафик из региона? Сколько времени уходит на это?

Впервые попробовав перенаправить трафик с помощью балансировщика нагрузки на основе DNS, вы поймете, насколько медленный этот процесс и как много разных механизмов, с помощью которых наши компьютеры кэшируют DNS-записи. Если применять DNS-шейпинг, то регион перестанет получать какие-либо запросы где-то через сутки. Делайте заметки независимо от того, как пройдет ваша первая попытка. Попробуйте определить, что сработало, а что — нет. Учитывая эти данные, поставьте перед собой цель — например, перенаправить 99 % трафика менее чем за 10 мин. Продолжайте практиковаться, пока данная цель не будет достигнута. Возможно, потребуются архитектурные изменения. Вам, наверное, придется автоматизировать этот процесс, чтобы команды не нужно было копировать и вставлять вручную. Независимо от необходимых изменений, постоянная практика поможет корректно реагировать на происшествия и находить участки архитектуры системы, требующие улучшения.

Подобного рода практику нужно проводить для любых действий, выполняемых в вашей системе. Отрабатывайте полномасштабное восстановление

и глобальный откат системы к предыдущей версии. Определите время, за которое вы бы хотели выполнять эти процедуры. Отмечайте все участки, в которых вы допускаете ошибки, и улучшайте их за счет дополнительных проверок и автоматизации. Если в ходе тренировок вы сможете достичь желаемого времени реакции, то будете уверены в том, что вам по силам должным образом отреагировать на реальное происшествие. Возьмите пример с пожарных и врачей скорой помощи, которые никогда не перестают тренироваться и учиться. Создайте план регулярных учений, чтобы каждый член вашей команды хорошо ориентировался в экстренных ситуациях, и, наверное, еще важнее, чтобы предусмотренные вами процедуры всегда оставались актуальными, несмотря на изменения в системе.

Рекомендации по глобальному развертыванию

- Распределите все свои образы по всему миру. Чтобы развертывание было успешным, ваш релиз (включая двоичные файлы, образы и т. д.) должен находиться рядом с тем местом, где его используют. Это также обеспечивает надежность развертывания в условиях перебоев в работе сети. Чтобы гарантировать согласованность, географическое распределение должно быть частью вашего автоматического процесса по созданию релизов.
- Старайтесь сделать так, чтобы как можно большая часть ваших тестов выполнялась на предварительных этапах; чем обширнее будет ваше интеграционное и нагрузочное тестирование, тем лучше. Развертывание следует начинать, только когда вы полностью уверены в его корректности.
- Начинайте развертывание с канареечного региона. Это среда, в которой другие команды или крупные клиенты могут убедиться в том, что *у них* не возникает никаких проблем при работе с вашим сервисом, прежде чем начнется полномасштабное развертывание.
- Определите отличительные особенности регионов, в которых происходит развертывание. Каждое отличие может стать причиной неполадок или полного/частичного отказа системы. Пытайтесь начинать развертывание с менее рискованных регионов.
- Документируйте и отрабатывайте процедуры (такие как откат на предыдущую версию) и реакцию на любые проблемы, с которыми можете столкнуться. Пытаясь вспомнить порядок действий в разгар кризисной ситуации, вы можете забыть о чем-то важном и только усугубить проблему.

Резюме

Это может показаться маловероятным, но большинству читателей данной книги рано или поздно придется управлять системами в глобальном масштабе. В этой главе мы попытались объяснить, как постепенно подготовить систему к глобальному развертыванию. Кроме того, мы обсудили меры, помогающие минимизировать время простоя в ходе обновления системы. В конце главы были рассмотрены процессы и процедуры, которые необходимо предусмотреть и отработать, чтобы быть готовыми к возможным (или, скорее, неизбежным) проблемам.

ГЛАВА 8

Управление ресурсами

В этой главе мы сосредоточимся на рекомендациях по администрированию и оптимизации ресурсов в Kubernetes. Мы обсудим планирование рабочих нагрузок, масштабирование приложений, а также управление кластером, pod и пространствами имен. Вы познакомитесь с некоторыми продвинутыми методами планирования, которые Kubernetes предоставляет через такие механизмы, как принадлежность и непринадлежность узлов (affinity, anti-affinity), ограничения (taints), допуски (tolerations) и селекторы узлов (nodeSelectors).

Мы покажем, как реализовать лимиты на ресурсы, запросы ресурсов, QoS pod, объекты `PodDisruptionBudget` и `LimitRanger`, а также политики непринадлежности.

Планировщик Kubernetes

Планировщик — один из важнейших компонентов Kubernetes, принадлежащих к управляющему уровню. Он определяет, на каком участке кластера развертываются pod, и обеспечивает оптимизацию ресурсов на основе ограничений кластера и пользовательских параметров. Планировщик использует алгоритм оценивания, основанный на предикатах и приоритетах.

Предикаты

Предикат — первая функция, с помощью которой Kubernetes принимает решение о том, на каких узлах может быть развернут тот или иной pod. Это подразумевает жесткое ограничение, вследствие чего функция возвращает либо `true`, либо `false`. В качестве примера приведем ситуацию, когда pod запрашивает 4 Гбайт памяти, а узел не может удовлетворить это требование. Узел вернет значение `false` и больше не будет считаться подходящим местом для развертывания pod. Точно так же не рассматриваются узлы, которые не предназначены для планирования.

Планировщик проверяет предикаты в порядке возрастания их сложности и жесткости накладываемых ими ограничений. На момент написания этой книги проверяются следующие предикаты:

```
CheckNodeConditionPred
CheckNodeUnschedulablePred
GeneralPred
HostNamePred
PodFitsHostPortsPred
MatchNodeSelectorPred
PodFitsResourcesPred
NoDiskConflictPred
PodToleratesNodeTaintsPred
PodToleratesNodeNoExecuteTaintsPred
CheckNodeLabelPresencePred
CheckServiceAffinityPred
MaxEBSVolumeCountPred
MaxGCEPDVolumeCountPred
MaxCSIVolumeCountPred
MaxAzureDiskVolumeCountPred
MaxCinderVolumeCountPred
CheckVolumeBindingPred
NoVolumeZoneConflictPred
CheckNodeMemoryPressurePred
CheckNodePIDPressurePred
CheckNodeDiskPressurePred
MatchInterPodAffinityPred
```

Приоритеты

Если предикаты однозначно определяют, подходит ли узел для планирования, приоритет позволяет сказать, насколько лучше подходит тот или другой узел. Ниже перечислены приоритеты, которые учитываются для узлов:

```
EqualPriority
MostRequestedPriority
RequestedToCapacityRatioPriority
SelectorSpreadPriority
ServiceSpreadingPriority
InterPodAffinityPriority
LeastRequestedPriority
BalancedResourceAllocation
NodePreferAvoidPodsPriority
NodeAffinityPriority
TaintTolerationPriority
ImageLocalityPriority
ResourceLimitsPriority
```

Оценки суммируются, и в конце узел получает итоговый приоритет. Например, если pod нужно 600 миллиядер, а в кластере доступно два узла, один с 900 миллиядрами, а другой с 1800, то последний будет иметь повышенный приоритет.

Если узлы получают одинаковый приоритет, то планировщик использует функцию `selectHost()`, которая делает выбор путем циклического перебора.

Продвинутые методики планирования

В большинстве ситуаций платформа Kubernetes хорошо справляется с оптимальным планированием работы pod. Она размещает pod только на узлах, имеющих достаточно ресурсов. Она также пытается распределить по разным узлам реплики из одного и того же набора ReplicaSet, чтобы повысить их доступность и сбалансировать потребление ресурсов. На случай, если этого недостаточно, Kubernetes позволяет вам вмешиваться в процесс планирования. Например, вы можете распределить pod по зонам доступности, чтобы ваше приложение могло продолжать работу в условиях отказа какой-либо зоны. Кроме того, некоторые pod имеет смысл размещать на одном сервере в целях повышения их производительности.

Принадлежность и непринадлежность pod

Принадлежность (affinity) и непринадлежность (anti-affinity) — механизмы, которые позволяют устанавливать правила размещения pod относительно друг друга. Эти правила влияют на процесс планирования и переопределяют решения, принимаемые планировщиком.

Например, с помощью правила непринадлежности реплики из ReplicaSet можно распределить по нескольким вычислительным центрам. Для этого pod назначаются метки. Если установить подходящую пару «ключ — значение», то планировщик может разместить pod на одном узле (то есть сделать так, чтобы они *принадлежали* к одному узлу) или предотвратить подобное размещение (сделать так, чтобы они *не принадлежали* к одному узлу).

Ниже приведен пример назначения pod правила непринадлежности:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
```

```
spec:
  selector:
    matchLabels:
      app: frontend
  replicas: 4
  template:
    metadata:
      labels:
        app: frontend
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - frontend
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: nginx
        image: nginx:alpine
```

В этом манифесте, предназначенном для развертывания NGINX, указаны четыре реплики и метка-селектор `app=frontend`. Кроме того, мы видим здесь правило PodAntiAffinity, которое гарантирует, что планировщик не станет размещать реплики на одном и том же узле. Таким образом, если некий узел выйдет из строя, то у вас останется достаточно реплик сервера NGINX для отдачи данных из его кэша.

nodeSelector

nodeSelector — самый простой способ разместить pod на определенном узле. Для принятия решений о планировании этот механизм использует метки-селекторы с парами «ключ — значение». Например, вы можете сделать так, чтобы ваши pod выполнялись на определенном узле со специальным оборудованием, например графическим адаптером. Вы могли бы спросить: «Разве то же самое нельзя сделать с помощью ограничения?» Действительно, это возможно. Но разница в том, что nodeSelector применяется, когда вы хотите *запросить* узел со специфической конфигурацией, а ограничение *резервирует* узел только для рабочих заданий, которым нужна эта конфигурация. Ограничения и селекторы nodeSelector можно использовать совместно,

чтобы, скажем, зарезервировать определенные узлы для задач, требующих наличия графического адаптера, и в то же время автоматически выбрать один из таких узлов.

Ниже показано, как назначить метку узлу и использовать nodeSelector в спецификации pod:

```
kubectl label node <имя_узла> disktype=ssd
```

Теперь создадим спецификацию с селектором `disktype: ssd`:

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
  labels:
    env: prod
spec:
  containers:
  - name: frontend
    image: nginx:alpine
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

В результате наш pod будет размещаться только на узлах с меткой `disktype=ssd`.

Ограничения и допуски

Ограничения (taints) назначаются узлам, чтобы на них нельзя было размещать определенные pod. Но ведь непринадлежность делает то же самое! Это правда, но ограничения используют другой подход и имеют другое назначение. Представьте, к примеру, что у вас есть pod, которым нужен определенный профиль производительности, и вы не хотите, чтобы на этих узлах выполнялись другие рабочие задания. Ограничения применяются совместно с *допусками* (tolerations), позволяющими переопределять ограниченные узлы. Сочетание этих двух механизмов позволяет гибко управлять правилами непринадлежности.

В целом ограничения и допуски используются, чтобы:

- выбирать узлы со специализированным оборудованием;
- выделять ресурсы узлов;
- избегать узлов с деградировавшей производительностью.

Существует несколько типов ограничений, влияющих на планирование и работу контейнеров:

- *NoSchedule* — жесткое ограничение, которое предотвращает развертывание на узле;
- *PreferNoSchedule* — размещает pod на данном узле, только если их нельзя развернуть где-то еще;
- *NoExecute* — «выселяет» с узла уже запущенные pod;
- *NodeCondition* — ограничение действует только для узлов, отвечающих заданному условию.

На рис. 8.1 показан пример узла с ограничением `gpu=true:NoSchedule`. У Pod 1 есть допуск `gpu`, поэтому он будет развернут на ограниченном узле. У Pod 2 есть допуск `no-gpu`, вследствие чего он не попадет на данный узел.

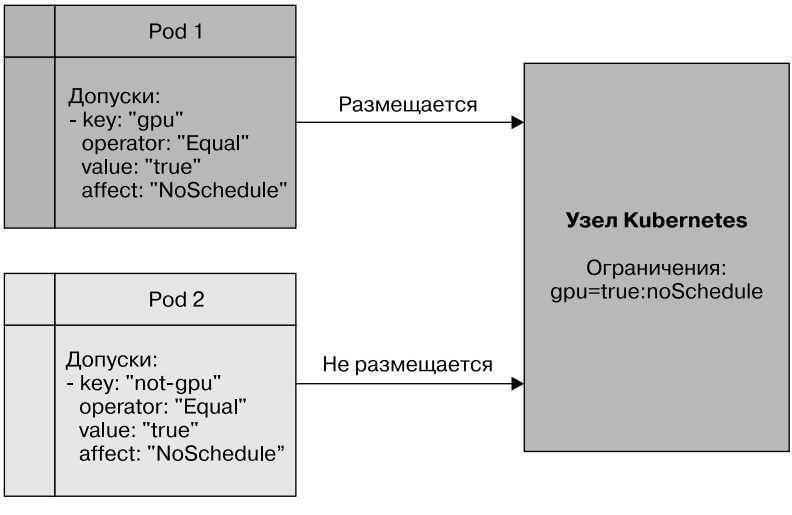


Рис. 8.1. Ограничения и допуски в Kubernetes

Если pod не удается развернуть из-за ограниченных узлов, то вы получите следующее сообщение об ошибке:

```
Warning: FailedScheduling 10s (x10 over 2m) default-scheduler 0/2 nodes are
available: 2 node(s) had taints that the pod did not tolerate.
```

Это пример того, как можно влиять на планирование с помощью ограничений, добавленных вручную. Но Kubernetes также поддерживает *выселе-*

ние активных pod *на основе ограничений* (taint-based eviction). Например, если узел перестает быть работоспособным из-за неисправного диска, то Kubernetes может разместить его pod на другом, исправном узле кластера.

Управление ресурсами pod

Один из самых важных аспектов администрирования приложений в Kubernetes заключается в корректном управлении ресурсами pod. Для этого нужно управлять процессорами и памятью, чтобы оптимизировать общий расход ресурсов в кластере. Ресурсами можно управлять на уровне как контейнера, так и пространства имен. Существуют и другие ресурсы, например сеть и хранилища данных, но Kubernetes пока еще не позволяет устанавливать для них запросы и лимиты.

Чтобы оптимизировать ресурсы и принимать разумные решения о размещении реплик, планировщик должен понимать требования приложения. Например, если приложению (контейнеру) нужно для работы как минимум 2 Гбайт памяти, то мы должны указать данный факт в спецификации его pod; в результате планировщик будет знать, сколько памяти должно быть доступно на узле, на котором будет развернут этот контейнер.

Запросы ресурсов

Запросы ресурсов в Kubernetes определяют, какое количество памяти или процессорных ядер нужно контейнеру. Если указать в спецификации pod запрос ресурсов в размере 8 Гбайт, а на всех ваших узлах свободно не больше 7,5 Гбайт памяти, то pod не будет развернут. В этом случае он будет находиться в состоянии *ожидания*, пока не освободятся нужные ему ресурсы.

Посмотрим, как это работает в нашем кластере.

Для начала определим, какие ресурсы нам доступны, используя команду `kubectl top`:

```
kubectl top nodes
```

Вывод должен выглядеть примерно так (у вашего кластера может быть другой объем памяти):

```
NAME                       CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
aks-nodepool1-14849087-0   524m         27%    7500Mi          33%
aks-nodepool1-14849087-1   468m         24%    3505Mi          27%
aks-nodepool1-14849087-2   406m         21%    3051Mi          24%
aks-nodepool1-14849087-3   441m         22%    2812Mi          22%
```

Как видите, максимальный объем памяти, доступный на отдельном сервере, равен 7500 МиБ, поэтому запланируем размещение pod, которому нужно 8000 МиБ:

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-request
spec:
  containers:
  - name: memory-request
    image: polinux/stress
    resources:
      requests:
        memory: "8000Mi"
```

Обратите внимание: pod будет оставаться в состоянии ожидания. Если просмотреть события, которые он генерирует, то можно увидеть, что для его размещения нет подходящих узлов:

```
kubectl describe pods memory-request
```

Вывод этой команды должен выглядеть так:

```
Events:
  Type     Reason            Age                From               Message
  Warning  FailedScheduling  27s (x2 over 27s)  default-scheduler  0/3 nodes
are available: 3 Insufficient memory.
```

Лимиты на ресурсы и качество обслуживания

Лимиты на ресурсы в Kubernetes позволяют указать максимальное количество процессорных ядер и памяти, которые выделяются pod. При достижении каждого из этих двух лимитов выполняется определенное действие. В случае с процессором контейнеру просто не дают использовать больше, чем указано в лимите. Если контейнер исчерпает выделенную ему память, то будет перезапущен; притом он может быть размещен как на том же, так и на другом узле кластера.

Назначив контейнеру лимиты, вы можете быть уверены в том, что приложения получат свою долю ресурсов:

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo
  namespace: cpu-example
spec:
  containers:
```

```
  - name: frontend
    image: nginx:alpine
    resources:
      limits:
        cpu: "1"
      requests:
        cpu: "0.5"

apiVersion: v1
kind: Pod
metadata:
  name: qos-demo
  namespace: qos-example
spec:
  containers:
  - name: qos-demo-ctr
    image: nginx:alpine
    resources:
      limits:
        memory: "200Mi"
        cpu: "700m"
      requests:
        memory: "200Mi"
        cpu: "700m"
```

Когда создается pod, ему назначается один из классов качества обслуживания (Quality of Service, QoS):

- гарантированный (guaranteed);
- взрывной (burstable);
- негарантированный (best effort).

Pod назначается *гарантированный* QoS-класс, если его запросы и лимиты совпадают (как для процессора, так и для памяти). *Взрывной* QoS-класс назначается, когда лимиты превышают запросы; это значит, что, помимо гарантированных ресурсов, контейнер может использовать дополнительные ресурсы, определяемые его лимитом. *Негарантированный* QoS-класс применяется, когда у контейнеров pod нет ни запросов, ни лимитов.

На рис. 8.2 показано, как pod назначаются QoS-классы.

Если вы хотите, чтобы все контейнеры в вашем pod имели гарантированное качество обслуживания, то должны установить для каждого из них лимиты и запросы (причем как для памяти, так и для процессора). В противном случае pod не будет назначен соответствующий QoS-класс.

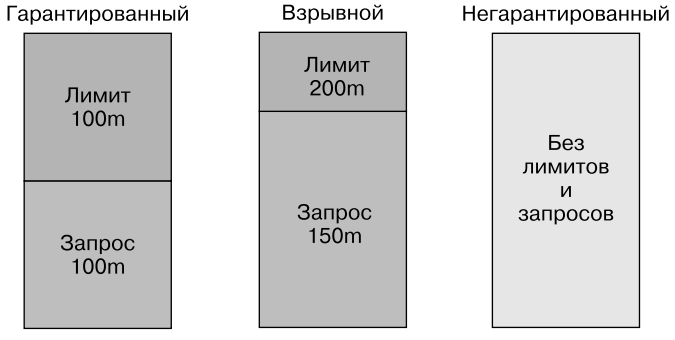


Рис. 8.2. Классы QoS в Kubernetes

Объекты PodDisruptionBudget

Однажды планировщику Kubernetes может понадобиться *выселить* (удалить) pod с узла. При этом их работа будет прервана либо *добровольно*, либо *принудительно*. Принудительное прерывание может быть вызвано аппаратными неполадками, недоступностью сети, сбоями в ядре или нехваткой ресурсов на узле. Причиной добровольного выселения может быть проведение технического обслуживания кластера, отключение лишних узлов в ходе автомасштабирования или обновление шаблонов pod. Минимизировать влияние этих процессов на ваше приложение вы можете с помощью объекта `PodDisruptionBudget`; он гарантирует, что приложение будет оставаться доступным даже во время выселения некоторых pod. Он позволяет указать минимальное количество доступных и максимальное количество недоступных реплик в ходе добровольного выселения. Пример добровольного выселения — ситуация, в которой узел освобождается от pod для проведения технического обслуживания.

Например, вы можете сделать так, чтобы в любой отдельно взятый момент времени недоступными могли быть не более 20 % pod вашего приложения. То же правило можно выразить в виде *X* реплик, которые всегда должны быть доступны.

Минимальное количество доступных реплик

В следующем примере мы задаем правило `PodDisruptionBudget`, согласно которому у frontend-приложения всегда должно быть не менее пяти доступных pod.

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: frontend-pdb
spec:
  minAvailable: 5
  selector:
    matchLabels:
      app: frontend
```

В этом примере `PodDisruptionBudget` описывает правило, согласно которому frontend-приложение всегда должно иметь не менее пяти реплик. Таким образом, Kubernetes может выселить любое количество pod при условии, что пять из них остаются доступными.

Максимальное количество недоступных реплик

В следующем примере мы задаем такое правило `PodDisruptionBudget`, чтобы у приложения frontend было не больше 20 % недоступных реплик:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: frontend-pdb
spec:
  maxUnavailable: 20%
  selector:
    matchLabels:
      app: frontend
```

Здесь указано, что в любой отдельно взятый момент времени недоступными могут быть не более 20 % pod. В этом сценарии во время добровольного прерывания работы может быть выселено максимум 20 % от всех реплик.

При проектировании кластера Kubernetes обязательно необходимо подумать об изменении его размера, чтобы он мог справиться с отказом какого-то количества узлов. Например, если у вас есть кластер с четырьмя узлами, один из которых выходит из строя, то вы теряете четверть своей производительности.

Процентные значения в PodDisruptionBudget могут не соотноситься с определенным количеством pod. Например, если у вашего приложения есть семь реплик и вы указали 50 % в поле maxAvailable, то не совсем понятно, какому количеству pod это соответствует: трем или четырем. В данном случае Kubernetes округляет значение до ближайшего целого числа, поэтому maxAvailable будет равно 4.

Управление ресурсами с помощью пространств имен

Пространство имен в Kubernetes обеспечивает удобное логическое разделение ресурсов, развернутых в кластере. Это позволяет задавать для каждого пространства имен свои квоты, правила RBAC (Role-Based Access Control — управление доступом на основе ролей) и сетевые политики. Таким образом ваш кластер поддерживает мягкую мультиарендность (multitenancy), благодаря которой вы можете разделять рабочие задания, не выделяя отдельную инфраструктуру команде или приложению. Это не только повышает эффективность использования ресурсов в кластере, но и обеспечивает логическое разделение.

Например, вы можете выделить каждой команде отдельное пространство имен и установить в нем квоту на доступные ресурсы, такие как процессор и память.

При проектировании пространств имен вам следует подумать о том, как вы хотите управлять доступом к определенному набору приложений. Если с вашим кластером будут работать разные команды, то каждой из них в идеале следует выделить по одному пространству имен. Если же ваш кластер предназначен только для одной команды, то отдельные пространства имен имеет смысл создавать для каждого развертываемого сервиса. Здесь нет какого-то универсального решения; архитектура кластера должна определяться организационной структурой вашей команды и обязанностями ее членов.

Сразу после развертывания ваш кластер Kubernetes будет содержать следующие пространства имен:

- `kube-system` — здесь развертываются внутренние компоненты Kubernetes, такие как `coredns`, `kube-proxy` и `metrics-server`;
- `default` — пространство имен, которое используется по умолчанию, если в спецификации объекта не указать поле `namespace`;
- `kube-public` — предназначено для анонимных и неаутентифицированных данных и зарезервировано для системного использования.

Пространство имен по умолчанию лучше не использовать, поскольку при управлении ресурсами очень легко наделать ошибок.

В ходе работы с пространствами имен с помощью утилиты `kubectl` следует указывать флаг `--namespace` (сокращенно `-n`):

```
kubectl create ns team-1

kubectl get pods --namespace team-1
```

Чтобы не добавлять флаг `--namespace` к каждой команде, утилите `kubectl` можно указать контекст с определенным пространством имен. Это легко сделать следующим образом:

```
kubectl config set-context my-context --namespace=team-1
```

При работе с несколькими пространствами имен и кластерами постоянное изменение контекста может быть довольно утомительным. Как показывает наш опыт, утилиты kubens (github.com/ahmetb/kubectx/blob/master/kubens) и kubectx (github.com/ahmetb/kubectx) могут помочь с переключением между разными пространствами имен и контекстами.

ResourceQuota

Когда в одном кластере работает несколько команд или приложений, для пространств имен необходимо предусмотреть квоты на ресурсы. Объект `ResourceQuota` позволяет выделить ресурсы кластера в логических единицах измерения и назначить каждому пространству имен их определенную долю.

Квоты можно устанавливать на следующие виды ресурсов.

- Вычислительные ресурсы:
  - `requests.cpu` — максимальное суммарное количество ядер, указанное в запросах;
  - `limits.cpu` — максимальное суммарное количество ядер, указанное в лимитах;
  - `requests.memory` — максимальный суммарный объем памяти, указанных в запросах;
  - `limits.memory` — максимальный суммарный объем памяти, указанный в лимитах.
- Ресурсы хранения данных:
  - `requests.storage` — максимальный суммарный объем хранилища, указанного в запросах;
  - `persistentvolumeclaims` — общее количество заявок на выделение `PersistentVolume`, которое может существовать в пространстве имен;
  - `storageclass.request` — максимальное количество заявок на выделение томов определенного класса;

- `storageclass.pvc` — общее количество заявок на выделение `PersistentVolume`, которое может существовать в пространстве имен для определенного `storageclass`.

❑ Квоты на количество объектов (неполный список):

- `count/pvc`;
- `count/services`;
- `count/deployments`;
- `count/replicasets`.

Как видите, Kubernetes поддерживает гибкое распределение квот на ресурсы между пространствами имен. Это повышает эффективность использования ресурсов в мультиарендных кластерах.

Теперь посмотрим, как эти квоты работают на практике. Примените следующий YAML-файл к пространству имен `team-1`:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
  namespace: team-1
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
    persistentvolumeclaims: "5"
    requests.storage: "10Gi

kubectl apply quota.yaml -n team-1
```

В этом примере пространству имен `team-1` устанавливаются квоты на процессор, память и хранилище.

Теперь посмотрим, как эти квоты повлияют на развертывание приложения:

```
kubectl run nginx-quotatest --image=nginx --restart=Never --replicas=1 --
port=80 --requests='cpu=500m,memory=4Gi' --limits='cpu=500m,memory=4Gi' -n
team-1
```

Процесс развертывания завершится неудачно из-за превышения квоты на память в размере `2Gi`:

```
Error from server (Forbidden): pods "nginx-quotatest" is forbidden: exceeded
quota: mem-cpu-demo
```

Как показывает этот пример, с помощью квот на ресурсы можно запретить развертывание приложений, не соответствующих правилам, которые вы задали для того или иного пространства имен.

LimitRange

Мы уже обсуждали установку запросов и лимитов на уровне контейнера, но что, если пользователь забудет указать эти параметры в спецификации pod? На этот случай Kubernetes предоставляет контроллер доступа, который установит `request` и `limits` автоматически.

Для начала создайте пространство имен для работы с квотами и `LimitRange`:

```
kubectl create ns team-1
```

Примените `LimitRange` к пространству имен, чтобы установить `defaultRequest` в разделе `limits`:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: team-1-limit-range
spec:
  limits:
  - default:
      memory: 512Mi
    defaultRequest:
      memory: 256Mi
    type: Container
```

Сохраните этот код в файле `limitranger.yaml` и выполните `kubectl apply`:

```
kubectl apply -f limitranger.yaml -n team-1
```

Убедитесь в том, что `LimitRange` применяет лимиты и запросы по умолчанию:

```
kubectl run team-1-pod --image=nginx -n team-1
```

Теперь запросите информацию о pod, чтобы увидеть, какие лимиты и запросы были установлены для ее спецификации:

```
kubectl describe pod team-1-pod -n team-1
```

Вы должны получить следующий результат:

```
Limits:
      memory:  512Mi
    Requests:
      memory:  256Mi
```

Квоты `ResourceQuota` необходимо использовать в сочетании с `LimitRange`, иначе любые объекты развертывания, в спецификации которых не указаны запросы или лимиты, будут отклоняться.

Масштабирование кластера

Одно из первых решений, которые необходимо принять при развертывании кластера, касается размера его узлов. Это больше похоже на искусство, чем на точную науку, особенно когда в одном и том же кластере выполняются разные рабочие задания. Вначале нужно выбрать подходящую отправную точку: пытайтесь соблюсти в одном узле баланс между мощностью процессора и объемом памяти. Определившись с характеристиками своего кластера, вы можете воспользоваться несколькими встроенными в Kubernetes механизмами, чтобы управлять его масштабированием.

Ручное масштабирование

Kubernetes упрощает масштабирование кластеров, особенно если вы используете такие инструменты, как Kops, или готовые решения на основе этой платформы. Для ручного масштабирования обычно достаточно указать нужное вам количество узлов, и Kubernetes самостоятельно добавит в ваш кластер новые серверы.

Вы также можете создавать пулы узлов, которые позволяют добавлять новые типы серверов в уже запущенный кластер. Это может быть крайне полезно, если в рамках одного кластера выполняются разные рабочие задания. Например, одним приложениям нужно больше процессорной мощности, а другим — памяти. С помощью пулов узлов в одном кластере можно сочетать разные типы серверов.

Но вам, наверное, не хочется делать это все вручную. На такой случай предусмотрено автомасштабирование, однако у данного механизма есть свои нюансы, и, как показывает наш опыт, большинству пользователей вначале лучше самостоятельно масштабировать свои узлы, заранее выделяя нужные ресурсы. Хотя, если ваши рабочие нагрузки сильно варьируются, автомасштабирование может быть очень кстати.

Автомасштабирование кластера

Kubernetes предоставляет дополнение Cluster Autoscaler, позволяющее указать минимальное и максимальное количество узлов, в пределах которого ваш кластер может масштабироваться. Решение о масштабировании прини-

мается в момент входа pod в состояние ожидания. Это, к примеру, происходит в ситуации, когда Kubernetes пытается развернуть pod, запрашивающий 4000 МиБ, а в кластере доступно только 2000 МиБ. Когда pod войдет в состояние ожидания, Cluster Autoscaler добавит для него новый узел и снова попытается его развернуть. Недостаток этого механизма состоит в том, что новый узел добавляется только после того, как pod входит в состояние ожидания, и пока это происходит, ваше рабочее задание будет простаивать. В Kubernetes v1.15 дополнение Cluster Autoscaler все еще не поддерживает масштабирование на основе пользовательских метрик.

Cluster Autoscaler может также уменьшать размер кластера при появлении у него ненужных ресурсов. Если в ресурсах больше нет необходимости, то лишний узел удаляется, а его pod распределяются по другим узлам кластера. Использование `PodDisruptionBudget` позволит избежать негативного воздействия на ваше приложение в ходе этого процесса.

Масштабирование приложений

Kubernetes предоставляет несколько механизмов для масштабирования приложений в кластере. Это можно делать вручную, изменяя количество реплик в спецификации `Deployment`. Вы также можете редактировать `ReplicaSet` или контроллер репликации, но мы не советуем управлять приложениями таким образом. Ручное масштабирование отлично подходит в ситуациях, когда ваши рабочие задания являются статическими или достаточно предсказуемыми. Но если вы испытываете внезапные скачки нагрузки или имеете дело с динамическими приложениями, то этот подход будет не самым оптимальным. К счастью, Kubernetes поддерживает автоматическое масштабирование с использованием HPA (Horizontal Pod Autoscaler — контроллер горизонтального автомасштабирования).

Сначала посмотрим, как вручную масштабировать объект `Deployment`, применяя следующий манифест:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 3
  template:
    metadata:
      name: frontend
      labels:
        app: frontend
```

```
spec:
  containers:
  - image: nginx:alpine
    name: frontend
    resources:
      requests:
        cpu: 100m
```

В этом примере развертывается три копии нашего клиентского сервиса. Мы можем изменить количество копий с помощью команды `kubectl scale`:

```
kubectl scale deployment frontend --replicas 5
```

В результате у нас получится пять реплик. Замечательно. Теперь попробуем автоматизировать этот процесс, чтобы приложение масштабировалось в зависимости от метрик.

Масштабирование с использованием HPA

Контроллер HPA позволяет масштабировать приложения на основе показателей процессора, памяти или пользовательских метрик. Он следит за объектом развертывания и запрашивает метрики у сервиса Kubernetes `metrics-server`. Кроме того, он позволяет указать минимальное и максимальное количество доступных pod. Например, вы можете определить политику HPA, согласно которой количество pod должно находиться в диапазоне от 3 до 10, а масштабирование должно происходить, когда загруженность процессора достигает 80 %. Устанавливать минимальное и максимальное значение следует всегда, иначе дефект в приложении может заставить контроллер HPA масштабировать реплики до бесконечности.

HPA поддерживает следующие параметры для синхронизации метрик и увеличения/уменьшения количества реплик:

- `horizontal-pod-autoscaler-sync-period` — по умолчанию метрики синхронизируются каждые 30 с;
- `horizontal-pod-autoscaler-upscale-delay` — по умолчанию между двумя операциями увеличения масштаба проходит 3 мин;
- `horizontal-pod-autoscaler-downscale-delay` — по умолчанию между двумя операциями уменьшения масштаба проходит 5 мин.

Указанные значения по умолчанию можно менять с помощью соответствующих флагов, но при этом нужно быть осторожными. Если ваша нагрузка колеблется слишком сильно, то вам стоит поэкспериментировать с разными параметрами, чтобы оптимизировать их для своих задач.

Определим политику HPA для приложения frontend, которое вы развернули в предыдущем упражнении.

Для начала сделайте приложение доступным на порте 80:

```
kubectl expose deployment frontend --port 80
```

Затем установите политику автомасштабирования:

```
kubectl autoscale deployment frontend --cpu-percent=50 --min=1 --max=10
```

Согласно этой политике ваше приложение будет иметь не меньше одной и не больше десяти реплик, а операция масштабирования инициируется, когда загруженность процессора достигнет 50 %.

Сгенерируем какой-нибудь трафик, чтобы посмотреть, как происходит автомасштабирование:

```
kubectl run -i --tty load-generator --image=busybox /bin/sh

Hit enter for command prompt
while true; do wget -q -O- http://frontend.default.svc.cluster.local; done

kubectl get hpa
```

Вам, возможно, придется подождать несколько минут, прежде чем реплики начнут масштабироваться автоматически.

С подробностями внутреннего устройства алгоритма автомасштабирования можно ознакомиться в проектном документе, доступном по адресу oreil.ly/nKnez.

HPA с применением пользовательских метрик

В главе 4 вы узнали, какую роль играет сервер метрик в мониторинге систем внутри Kubernetes. API данного сервера также позволяет масштабировать приложения с учетом пользовательских метрик. В сочетании с Metrics Aggregator это дает сторонним провайдерам возможность подключать и расширять метрики, на основе которых HPA может принимать решения о масштабировании. Например, приложения можно масштабировать не только с учетом загруженности процессора и памяти, но и в зависимости от показателей, предоставляемых сервисом очередей. Это позволяет выполнять автоматическое масштабирование с применением метрик, относящихся к конкретному приложению или взятых из внешнего сервиса.

Масштабирование с использованием VPA

VPA (Vertical Pod Autoscaler — контроллер вертикального автомасштабирования pod) отличается от HPA тем, что масштабирует не реплики, а запросы. Ранее в главе мы уже обсуждали установку запросов для наших pod и то, как это позволяет гарантированно выделить определенное количество ресурсов для заданного контейнера. Благодаря VPA эти запросы можно больше не регулировать вручную — они автоматически масштабируются в обоих направлениях. Этот механизм хорошо подходит для рабочих заданий, которые не могут масштабироваться ввиду своей архитектуры. Например, база данных MySQL масштабируется не так, как клиентское веб-приложение, но вы можете настроить автоматическое изменение запрошенных ресурсов ее ведущих узлов в зависимости от нагрузки.

VPA отличается от HPA повышенной сложностью и состоит из трех компонентов:

- `Recommender` — отслеживает текущее и предыдущее потребление ресурсов, предоставляя рекомендуемые значения для запросов процессора и памяти в контейнере;
- `Updater` — удаляет pod, для которых установлены неадекватные ресурсы, чтобы их контроллеры могли создать их заново с обновленными запросами ресурсов;
- `Admission Plugin` — устанавливает подходящие запросы ресурсов для новых pod.

На момент выхода Kubernetes v1.15 контроллер VPA не рекомендуется использовать для промышленного развертывания.

Рекомендации по управлению ресурсами

- Используйте механизм непринадлежности pod для распределения рабочих заданий между несколькими зонами доступности. Это сделает ваше приложение высокодоступным.
- Если вы применяете специальное оборудование, такое как графические адаптеры, то убедитесь, что оно доступно только тем рабочим заданиям, которым оно действительно нужно. Применяйте для этого механизм ограничений.
- Используйте ограничения `NodeCondition`, чтобы предотвратить отказ или деградацию узлов.

- Применяйте селекторы `nodeSelector` к спецификациям своих pod, чтобы развертывать их на узлах со специальным оборудованием.
- Прежде чем развертывать систему в промышленной среде, попробуйте подобрать оптимальный размер узлов с точки зрения денежных расходов и производительности.
- Если вы развертываете рабочие задания разных типов с разными характеристиками производительности, то используйте пулы узлов, чтобы сочетать разные типы серверов в одном кластере.
- Убедитесь, что вы установили ограничения на объем памяти и процессора для всех pod, развернутых в вашем кластере.
- Используйте объекты `ResourceQuota` для выделения разным командам или приложениям причитающейся им доли ресурсов кластера.
- Создавайте объекты `LimitRange`, чтобы устанавливать лимиты и запросы по умолчанию для спецификаций pod, не имеющих этих параметров.
- Сначала масштабируйте свой кластер Kubernetes вручную, пока не разберетесь с тем, какого рода нагрузки он испытывает. Вы также можете использовать автомасштабирование, но при этом помните о нюансах, связанных с временем запуска узлов и уменьшением масштаба кластера.
- Применяйте HPA для рабочих динамических заданий, которые могут испытывать внезапные скачки нагрузки.

Резюме

В данной главе мы обсудили, как оптимизировать управление ресурсами кластера и приложений. Kubernetes предоставляет для этого множество встроенных средств, которые помогут сделать вашу систему надежной, высоконагруженной и эффективной. Изменение размера кластера и количества pod может быть непростой задачей, особенно вначале, но отслеживая работу своих приложений в промышленных условиях, вы сможете найти способы оптимизации ресурсов.

ГЛАВА 9

Сетевые возможности, безопасность сети и межсервисное взаимодействие

Kubernetes фактически является диспетчером систем, распределенных по кластеру, но связанных между собой. Огромное значение имеет то, как эти системы общаются друг с другом, и ключевую роль здесь играет сеть. Понимание того, как Kubernetes обеспечивает совместную работу своих распределенных сервисов, необходимо для эффективного применения средств межсервисного взаимодействия.

Текущая глава посвящена принципам работы с сетью, заложенным в Kubernetes, и рекомендациям относительно применения этих принципов в разных ситуациях. Сеть и вопросы безопасности обычно идут рука об руку. Традиционные модели сетевой безопасности, в которых доступ контролируется на уровне самой сети, имеют место и в нашем новом мире распределенных систем, но то, как их реализуют и какие возможности они предлагают, претерпело небольшие изменения. У Kubernetes есть встроенный API для работы с политиками сетевой безопасности, подозрительно похожий на старый добрый брандмауэр и его правила.

В последнем разделе главы мы погрузимся в новый и пугающий мир механизмов межсервисного взаимодействия. Конечно, слово «пугающий» использовано в шутку, но в Kubernetes эта технология довольно молода и мало освоена.

Принципы работы с сетью в Kubernetes

Понимать, как Kubernetes использует сетевые возможности для организации взаимодействия сервисов, необходимо для эффективного проектирования приложений. Темы, связанные с сетью, вызывают серьезную головную боль у большинства людей. Мы постараемся ничего не усложнять, поскольку

это скорее набор рекомендуемых подходов, а не урок по сетевым возможностям контейнеров. К счастью для нас, в Kubernetes уже существуют общие правила и ориентиры, с которых мы и начнем. Речь идет о том, как должны общаться между собой разные компоненты. Подробно рассмотрим каждое из этих правил.

- *Взаимодействие контейнеров в одном pod.* Все контейнеры одного pod находятся в общем сетевом пространстве. Помимо прочего, это означает, что они должны открывать разные порты. Для этого в каждом pod находится неактивный контейнер, предназначенный сугубо для размещения сетевых инструментов. В сочетании с пространствами имен Linux и сетевыми возможностями Docker это позволяет контейнерам работать в одной и той же локальной сети. На рис. 9.1 показано, как контейнер A обращается напрямую к контейнеру B, используя localhost и номер порта, который тот прослушивает.

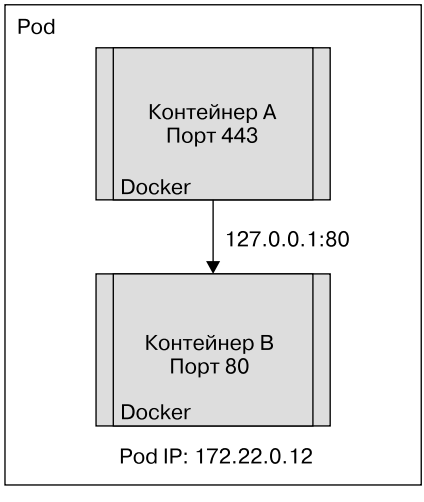


Рис. 9.1. Взаимодействие контейнеров в одном pod

- *Взаимодействие pod.* Pod нужно общаться между собой без преобразования сетевых адресов (network address translation, NAT). То есть IP-адрес, который видит принимающий pod, действительно принадлежит отправителю. Это можно организовать разными способами, в зависимости от сетевых дополнений, использующихся в кластере, но об этом мы поговорим чуть позже. Данный принцип распространяется на все pod, независимо от того, находятся ли они на одном узле. Это также подразумевает, что

узел должен общаться с pod напрямую, без использования NAT. Благодаря этому агенты и системные демоны, размещенные на узлах, тоже могут при необходимости обращаться к pod. На рис. 9.2 показан процесс взаимодействия pod на одном узле и на разных узлах кластера.

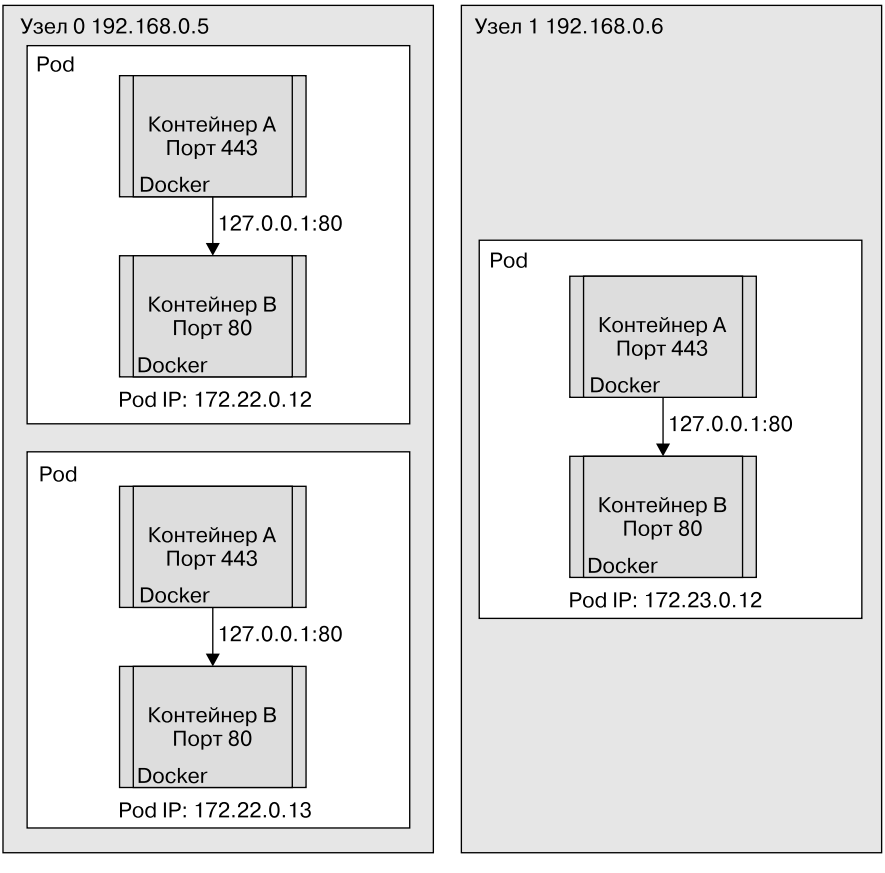


Рис. 9.2. Взаимодействие pod на одном узле и на разных узлах

- *Взаимодействие сервисов и pod.* Сервисы в Kubernetes представляют собой постоянные IP-адрес и порт, которые перенаправляют трафик к связанным с ними конечным точкам. В разных версиях Kubernetes это реализовывалось по-разному, однако на сегодня для этого используется два основных метода: iptables и более новый подход на основе IPVS (IP Virtual Server). Большинство современных реализаций применяют iptables для создания на каждом узле балансировщика нагрузки поверх псевдотранспортного

уровня (pseudo-Layer 4). На рис. 9.3 показан принцип связывания сервиса и pod через метки-селекторы.

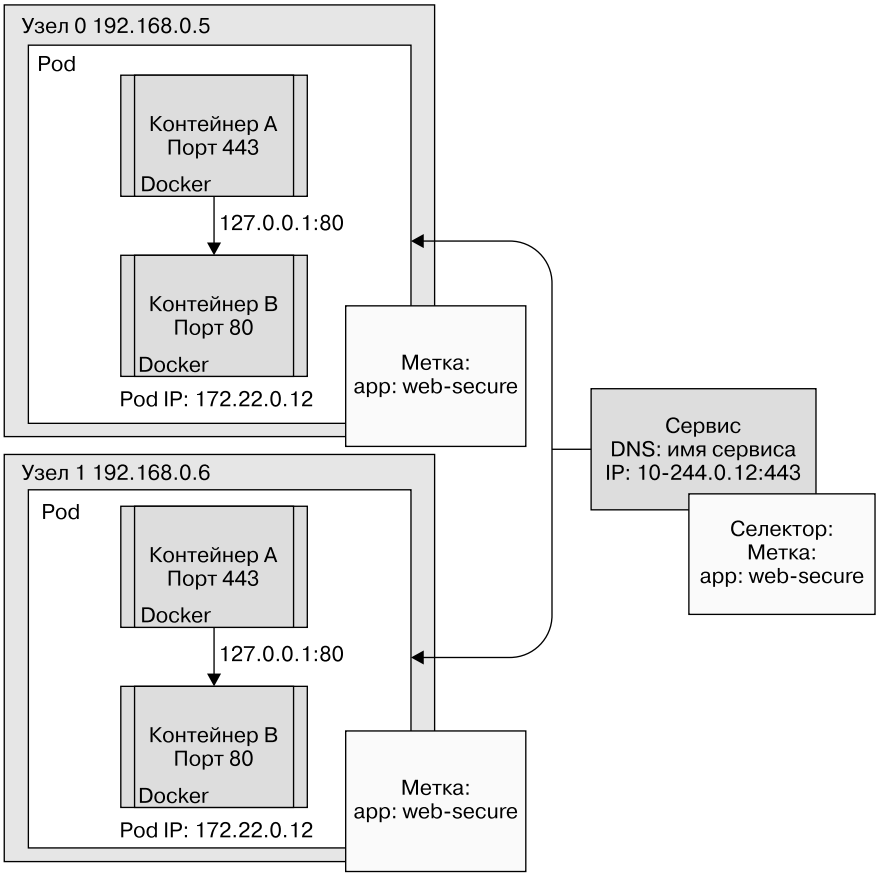


Рис. 9.3. Взаимодействие сервиса и pod

Сетевые дополнения

На ранних этапах становления Интернета развитием сетевых стандартов занималось сообщество Special Interest Group (SIG). Благодаря этому мы имеем модульную архитектуру, давшую дорогу большому количеству сторонних сетевых проектов, многие из которых позволили расширить возможности рабочих заданий в Kubernetes. Эти сетевые дополнения делятся на две категории. К первой из них можно причислить наиболее простое дополнение

под названием Kubenet, которое является стандартным для Kubernetes. Ко второй категории относится спецификация Container Network Interface (CNI), представляющая собой универсальное решение для подключения сетевых модулей к контейнерам.

Kubenet

Kubenet — простейшее сетевое дополнение, входящее в стандартную поставку Kubernetes. Оно предоставляет сетевой мост в Linux, `cbr0`, который создает Ethernet-соединение между pod, подключенными к нему. Pod получает IP-адрес из диапазона бесклассовой адресации (Classless Inter-Domain Routing, CIDR), распределенного между узлами кластера. Чтобы к трафику, направленному к IP-адресам за пределами данного диапазона, можно было применять маскарадинг (NAT overload), устанавливается специальный флаг. Это соответствует правилам взаимодействия pod, поскольку через NAT проходит только тот трафик, который направлен за пределы диапазона CIDR pod. Пакет, направленный от одного узла к другому, проходит через некий механизм маршрутизации.

Рекомендации по использованию Kubenet

- Kubenet реализует упрощенный сетевой стек и не занимает драгоценные IP-адреса в уже заполненных сетях. Это особенно актуально для облачных сетей, расширяемых за счет локальных вычислительных центров.
- Убедитесь в том, что диапазон CIDR способен вместить ваши кластеры со всеми их pod. По умолчанию kubenet позволяет размещать не больше 110 pod на каждом узле, но этот показатель можно откорректировать.
- Корректно планируйте правила маршрутизации, чтобы трафик направлялся к pod на подходящих узлах. В облачных сервисах данный механизм, как правило, реализуется самим провайдером, но при локальном размещении и в крайних случаях может потребовать автоматизации и надежного сетевого администрирования.

Дополнение CNI

Спецификация CNI устанавливает ряд основных требований к дополнениям, создаваемым на ее основе. В их число входят определенные интерфейсы и API-вызовы, с помощью которых дополнение должно взаимодействовать

со средой выполнения контейнера, применяемой в кластере. CNI также описывает элементы сетевого администрирования, такие как управление IP-адресами и возможность добавлять и удалять контейнеры из сети. Полный текст спецификации, который изначально был основан на проекте `rkt`, доступен на странице https://oreil.ly/wGvF7.

Проект Core CNI предоставляет библиотеки для написания дополнений, которые удовлетворяют основным требованиям спецификации и способны взаимодействовать с другими дополнениями в целях реализации разного рода возможностей. Благодаря такой гибкости мы имеем целый ряд дополнений CNI, применимых для сетевого администрирования контейнеров в таких облаках, как Microsoft Azure (встроенное дополнение CNI) и Amazon Web Services (дополнение VPC CNI), а также в традиционных сетевых решениях, таких как Nuage CNI, Juniper Networks Contrail/Tunsten Fabric и VMware NSX.

Рекомендации по использованию CNI

Сетевые возможности — один из ключевых элементов исправно работающей среды Kubernetes. Взаимодействие внутренних виртуальных компонентов Kubernetes и физических сетевых механизмов должно быть тщательно спроектировано, чтобы обеспечить надежный обмен данными с приложением.

1. Подумайте, какой набор возможностей необходим для реализации нужной вам сетевой инфраструктуры. Некоторые дополнения CNI имеют встроенную поддержку высокой доступности, установления сетевых соединений между разными облаками, сетевых политик Kubernetes и др.
2. Если вы размещаете свои кластеры в публичных облаках, то убедитесь, что ваши дополнения CNI поддерживаются в программно-определяемой сети (Software-Defined Network, SDN) вашего облачного провайдера.
3. Проверьте, совместимо ли ваше дополнение CNI со средствами сетевой безопасности и наблюдаемости, а также инструментами сетевого администрирования; если нет, то попробуйте найти им замену. Очень важно сохранить безопасность и наблюдаемость сети, поскольку потребность в этих возможностях будет расти с переходом на крупномасштабные распределенные системы, такие как Kubernetes. В любую среду можно интегрировать решения, подобные Weaveworks Weave Scope, Dynatrace

и Sysdig, каждое из которых имеет свои преимущества. Если вы используете облачную версию Kubernetes, такую как Azure AKS, Google GCE или AWS EKS, то пытайтесь применять ее стандартные инструменты: Azure Container Insights and Network Watcher, Google Stackdriver, AWS CloudWatch и т. д. Любое из этих средств должно как минимум предоставлять информацию о сетевом стеке и четыре золотых сигнала, которые стали популярными благодаря феноменальной команде Google SRE и Робу Иващуку: латентность, трафик, ошибки и степень загруженности.

4. Если ваши дополнения CNI не предоставляют оверлейную сеть, отделенную от диапазона SDN, то убедитесь, что вашего адресного сетевого пространства хватит для всех узлов, pod и внутренних балансировщиков нагрузки, а также для потенциального обновления и масштабирования кластера.

Сервисы в Kubernetes

Благодаря базовым правилам организации сети в Kubernetes и сетевым дополнениям, реализующим эти правила, развертываемые pod могут взаимодействовать только внутри одного кластера. Некоторые дополнения CNI назначают pod IP-адреса в одном адресном пространстве с узлами, поэтому теоретически, зная данный IP, вы можете подключиться к pod напрямую из-за пределов Kubernetes. Но это не самый эффективный способ доступа к сервисам, поскольку pod в Kubernetes по своей природе непостоянны. Представьте, что у вас есть функция или система, которой нужно обратиться к API, доступному в pod. Какое-то время схема может работать нормально, однако рано или поздно возникнет добровольное или принудительное прерывание, в результате которого данный pod исчезнет. Kubernetes, вероятно, создаст ему замену с новыми именем и IP-адресом, поэтому очевидно, что для его поиска нужен некий механизм. Здесь вам пригодится API для работы с сервисами.

Данный API позволяет назначать конечным точкам сервиса постоянные IP-адрес и порт в пределах кластера и автоматически привязывать их к подходящим pod. Этот магический процесс основан на упомянутых выше механизмах iptables и IPVS, работающих на уровне узлов Linux; он привязывает IP-адрес и порт сервиса к реальным IP-адресам конечной точки или pod. Контроллер, который отвечает за это, называется `kube-proxy`; он выполняется на каждом узле кластера, управляя правилами iptables.

При определении объекта `Service` необходимо указать тип сервиса. От этого зависит, где будет доступна конечная точка: только внутри кластера или за его пределами. Существует четыре основных типа сервисов, которые мы кратко обсудим в следующих подразделах.

Тип сервисов ClusterIP

Если не указать в спецификации сервиса его тип, то по умолчанию будет использоваться ClusterIP. Сервисам данного типа назначаются IP-адреса из выделенного диапазона CIDR. Эти адреса действительны на протяжении всего жизненного цикла сервисов и привязываются к IP-адресам и портам клиентских pod с помощью поля `selector`. Но, как вы увидите сами, в некоторых случаях селектор использовать нельзя. В спецификации сервиса также указывается его доменное имя. На этом основан механизм обнаружения внутри кластера, позволяющий рабочим заданиям с легкостью обращаться к другим сервисам в том же кластере, используя DNS-запросы. Например, если у вас есть спецификация сервиса, представленная ниже, и вам нужно сделать HTTP-вызов к этому сервису из другого pod внутри того же кластера, то вы можете использовать адрес http://web1-svc (при условии, что клиент и сервис находятся в одном пространстве имен):

```
apiVersion: v1
kind: Service
metadata:
  name: web1-svc
spec:
  selector:
    app: web1
ports:
- port: 80
  targetPort: 8081
```

Если вам нужен доступ к сервисам в других пространствах имен, то вы можете использовать доменные имена вида `<имя_сервиса>.<название_пространства_имен>.svc.cluster.local`.

При отсутствии в определении сервиса селектора для него можно определить API с конечными точками. В результате сервис будет доступен по заданным IP-адресу и порту, и для этого не требуется атрибут `selector`, который автоматически обновляет конечные точки из pod, соответствующих селектору. Это может пригодиться в ряде ситуаций, если у вас есть база данных, предназначенная для тестирования, но находящаяся за пределами кластера, и вы хотите, чтобы ваш сервис использовал вместо нее другую БД, развернутую

в Kubernetes. Такие сервисы иногда называют *неуправляемыми* (headless), поскольку они, в отличие от обычных, не управляются контроллером `kube-proxy`, но позволяют работать с конечными точками напрямую, как показано на рис. 9.4.

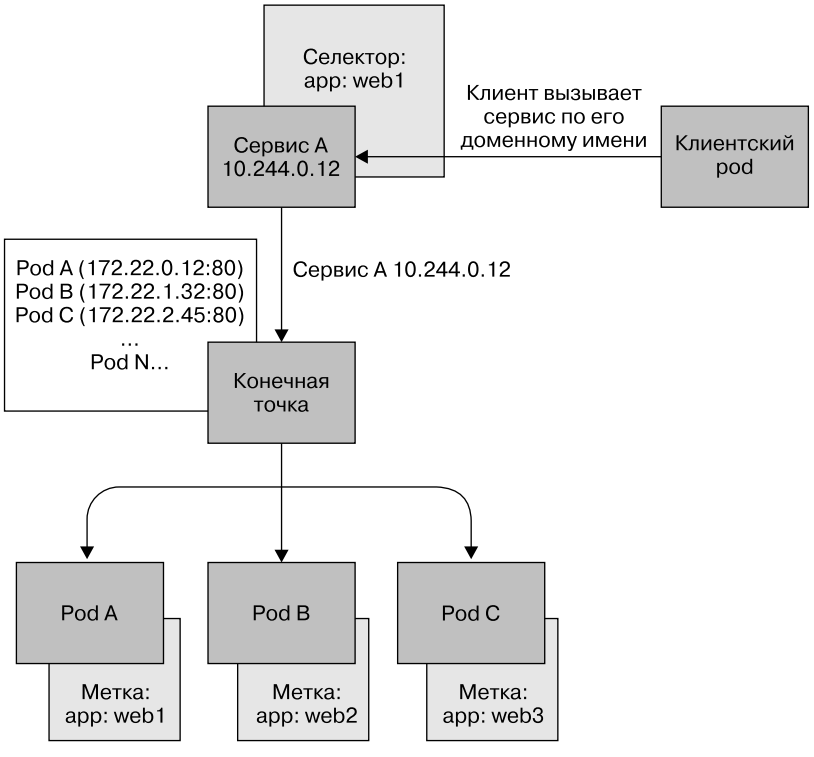


Рис. 9.4. Принцип работы сервисов ClusterIP

Тип сервисов NodePort

Сервисы типа NodePort привязывают свои IP-адреса и порты к портам высшего уровня на каждом узле кластера. Порты высшего уровня находятся в диапазоне от 30 000 до 32 767 и могут быть либо динамически назначены, либо явно заданы в спецификации сервиса. Они обычно используются в кластерах с локальным размещением или в самописных решениях, которые не поддерживают конфигурацию с автоматической балансировкой нагрузки. Для прямого доступа к сервису из-за пределов кластера используйте адрес вида `<IP_узла>:<порт_узла>`, как показано на рис. 9.5.

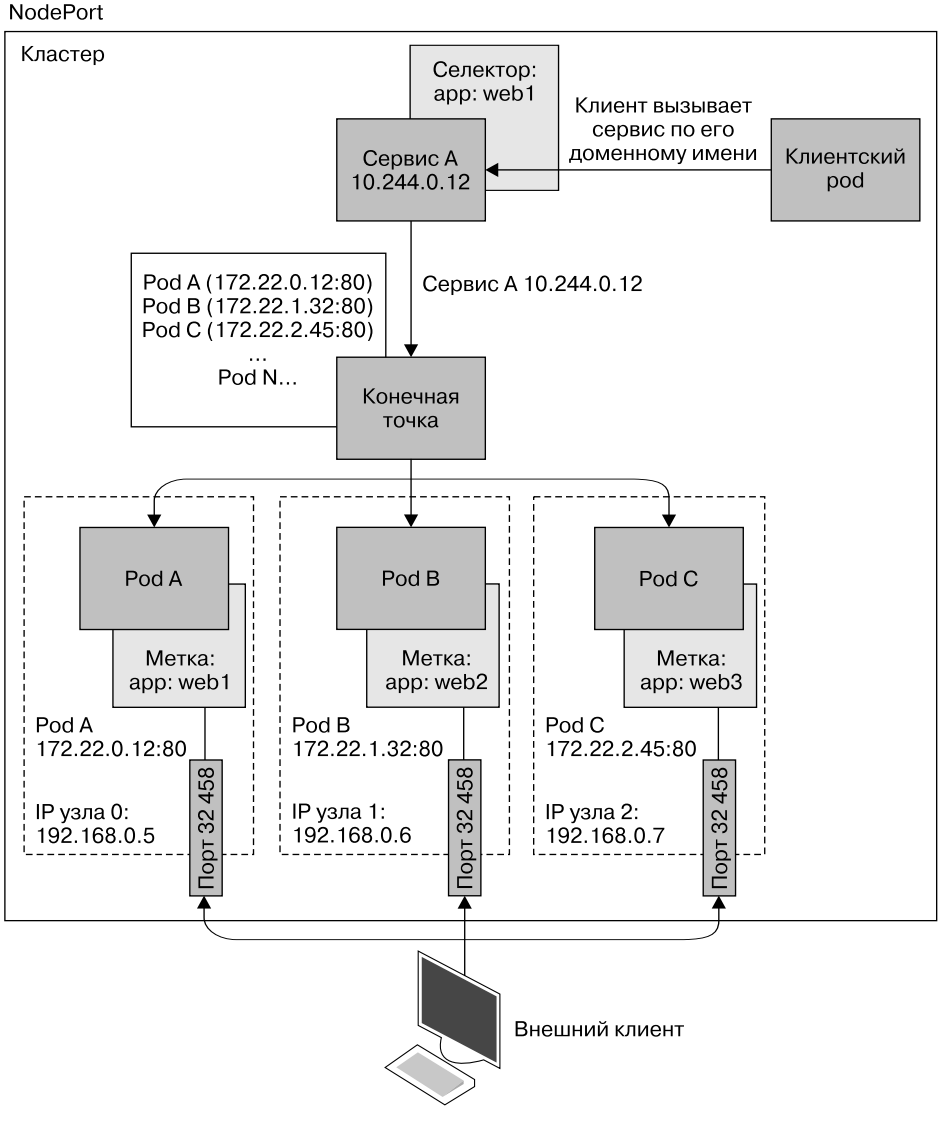


Рис. 9.5. Организация pod, сервисов и сети с помощью NodePort

Тип сервисов ExternalName

Сервисы типа `ExternalName` редко применяются на практике, но могут пригодиться для привязки постоянных внутренних доменных имен кластера к сервисам с внешними доменными именами. Типичный пример — внешняя

база данных облачного провайдера с уникальным доменным именем, таким как `mymongodb.documents.azure.com`.

Строго говоря, это можно легко прописать в спецификации pod, используя переменную `Environment` (см. главу 6). Но лучше задействовать более общее внутреннее имя, такое как `prod-mongodb`. что позволит изменить базу данных, на которую оно ссылается, простым редактированием спецификации сервиса. В случае изменения переменной `Environment` вам пришлось бы перезапускать pod.

```
kind: Service
apiVersion: v1
metadata:
  name: prod-mongodb
  namespace: prod
spec:
  type: ExternalName
  externalName: mymongodb.documents.azure.com
```

Тип сервисов LoadBalancer

`LoadBalancer` — специальный тип сервисов, позволяющих автоматизировать взаимодействие с облачными провайдерами и другими программируемыми сервисами облачной инфраструктуры. С их помощью можно развернуть механизм балансировки нагрузки, предоставляемый облаком, в котором размещен кластер Kubernetes. Это означает, что в большинстве случаев сервис `LoadBalancer` будет работать примерно так же, как публично доступный балансировщик нагрузки от AWS, Azure, GCE, OpenStack и т. д. Однако каждый облачный провайдер предоставляет собственные аннотации для поддержки дополнительных возможностей, таких как сугубо внутреннее распределение нагрузки, установка параметров конфигурации для AWS ELB и т. д. В спецификации сервиса можно вдобавок указать IP-адрес балансировщика и допустимые диапазоны. Пример этого показан в следующем листинге и на рис. 9.6.

```
kind: Service
apiVersion: v1
metadata:
  name: web-svc
spec:
  type: LoadBalancer
  selector:
    app: web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8081
```

```
loadBalancerIP: 13.12.21.31
loadBalancerSourceRanges:
- "142.43.0.0/16"
```

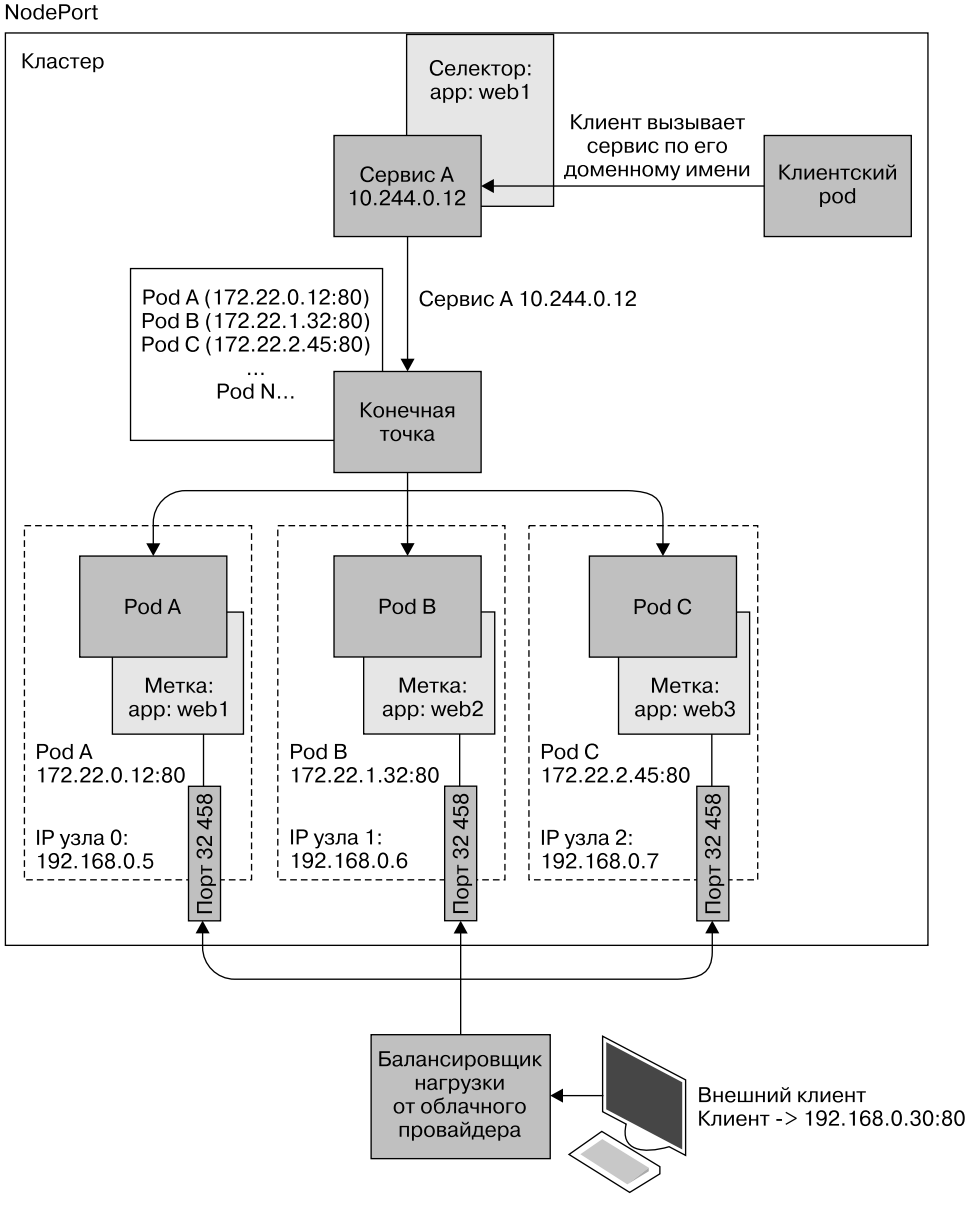


Рис. 9.6. Организация pod, сервисов, узлов и сети с помощью LoadBalancer

Объекты и контроллеры Ingress

Формально спецификация Ingress не считается типом сервиса, но это важный механизм для направления трафика к рабочим заданиям в Kubernetes. API сервисов поддерживает базовое распределение нагрузки на транспортном и сетевом уровнях (Layer 3/4). Но в реальности многие сервисы, которые развертываются в Kubernetes и не хранят свое состояние, нуждаются в управлении трафиком на более высоком, прикладном уровне — в частности, на уровне протокола HTTP.

API Ingress, по сути, представляет собой HTTP-маршрутизатор, позволяющий направлять запросы к отдельным клиентским сервисам с помощью правил на основе доменных имен и путей. Представьте, что сайт с доменным именем www.evillgenius.com имеет два пути, /registration и /labaccess, обслуживающиеся двумя разными сервисами, развернутыми в Kubernetes, `reg-svc` и `labaccess-svc`. Вы можете определить правило доступа, согласно которому запросы к www.evillgenius/registration будут передаваться сервису `reg-svc` и соответствующим pod; точно так же подходящие конечные точки сервиса `labaccess-svc` могут получать запросы, направленные к www.evillgenius.com/labaccess. API Ingress также поддерживает маршрутизацию на уровне узлов, позволяя указывать разные серверы в одном и том же правиле доступа. Кроме того, вы можете создать объект `Secret` с информацией о сертификатах для работы с TLS на порте 443. На случай, если путь не указан, обычно предусматривают обработчик по умолчанию, который возвращает нечто более полезное, чем стандартная страница с ошибкой 404.

Деталями конфигурации конкретных TLS-путей и обработчика по умолчанию занимается так называемый контроллер Ingress. Этот контроллер отделен от API Ingress и позволяет системным администраторам развертывать разные реализации, такие как NGINX, Traefik, HAProxy и т. д. В отличие от других контроллеров Kubernetes, он не является частью системы; это сторонний механизм, который работает с динамической конфигурацией через API Ingress. Самая распространенная реализация контроллера Ingress, NGINX, частично поддерживается проектом Kubernetes, но имеет множество аналогов, как открытых, так и коммерческих:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: labs-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
```

```
spec:
  tls:
  - hosts:
    - www.evillgenius.com
    secretName: secret-tls
rules:
- host: www.evillgenius.com
  http:
    paths:
    - path: /registration
      backend:
        serviceName: reg-svc
        servicePort: 8088
    - path: /labaccess
      backend:
        serviceName: labaccess-svc
        servicePort: 8089
```

Рекомендации по использованию сервисов и контроллеров Ingress

Создание сложных виртуальных сетей со связанными между собой приложениями требует тщательного планирования. Для эффективного управления взаимодействием разных сервисов друг с другом и с внешним миром необходимо постоянное внимание к изменениям, происходящим в приложении. Облегчить этот процесс помогут следующие рекомендации.

- Ограничьте количество сервисов, к которым нужно обращаться из-за пределов кластера. В идеале большинство сервисов должны иметь тип ClusterIP, а доступными снаружи должны быть только сервисы, действительно нуждающиеся в этом.
- Если сервисы, которые должны быть доступны снаружи, работают преимущественно по HTTP/HTTPS, то лучше всего использовать API и контроллер Ingress в сочетании с прокси-сервером для поддержки TLS. Некоторые типы контроллеров Ingress имеют встроенную поддержку ограничения частоты запросов, переопределения заголовков, аутентификации OAuth, наблюдаемости и прочих возможностей, которые можно не реализовывать в самих приложениях.
- Выберите контроллер Ingress с поддержкой возможностей, необходимых для безопасного управления трафиком ваших рабочих заданий. Выберите какую-то одну реализацию и используйте ее во всех своих промышленных средах, поскольку разные контроллеры предоставляют специфические

аннотации, которые затрудняют перемещение кода между кластерами Kubernetes.

- Исследуйте возможности контроллера Ingress, который предоставляет ваш облачный провайдер, и попробуйте вынести инфраструктуру, необходимую для управления трафиком, за пределы кластера. Но убедитесь, что по-прежнему можете управлять этой инфраструктурой с помощью конфигурационных файлов Kubernetes.
- Если ваши API в основном доступны снаружи, то обратите внимание на специализированные контроллеры Ingress, такие как Kong и Ambassador. Они поддерживают тонкую настройку рабочих заданий, основанных на API. NGINX; Traefik и другие решения тоже позволяют конфигурировать API, но являются не столь гибкими по сравнению со специализированными прокси-серверами.
- При развертывании контроллеров Ingress в Kubernetes в виде рабочих заданий на основе pod убедитесь, что они высокодоступны, и агрегируйте показатели их производительности. В целях обеспечения их адекватного масштабирования используйте метрики, но в то же время не забудьте предусмотреть резервные ресурсы, чтобы не терять соединения с клиентами при изменении масштаба.

Сетевые политики безопасности

API `NetworkPolicy`, встроенный в Kubernetes, позволяет управлять входящим и исходящим трафиком ваших рабочих заданий на уровне сети. С помощью сетевых политик можно определить, каким группам pod позволено взаимодействовать друг с другом и с иными конечными точками. Более глубокие аспекты спецификации `NetworkPolicy` могут показаться запутанными, особенно учитывая то, что они описаны в виде API Kubernetes. При этом вам нужно сетевое дополнение, которое поддерживает API `NetworkPolicy`.

Сетевые политики описываются в формате YAML и на первый взгляд могут показаться довольно сложными. Можно понимать их лучше, если представить, что это правила для обычного брандмауэра внутри вычислительного центра. Спецификация каждой политики содержит поля `podSelector`, `ingress`, `egress` и `policyType`, обязательным из которых является лишь первое. Поле `podSelector` работает по тому же принципу, что и селекторы Kubernetes с метками `matchLabels`. Вы можете создавать разные определения

NetworkPolicy для одних и тех же pod, и они будут дополнять друг друга. Поскольку объекты NetworkPolicy действуют на уровне пространства имен, в случае отсутствия поля podSelector они применяются ко всем pod этого пространства. На основе правил для входящего и исходящего трафика фактически формируется белый список, который определяет, что может принимать и отправлять pod. Но здесь есть одно важное отличие: если в результате применения селектора pod попадает в область действия сетевой политики, весь его трафик, кроме указанного во входящих и исходящих правилах, блокируется. Этот небольшой нюанс означает, что pod, на которые не распространяются сетевые политики, могут принимать и отправлять любые запросы. Это было сделано специально, во избежание блокировки новых рабочих заданий, развертывающихся в Kubernetes.

Поля ingress и egress — это, в сущности, список правил, основанных на источнике или месте назначения запроса; они могут содержать определенные диапазоны CIDR, а также селекторы podSelector и namespaceSelector. Если оставить поле ingress пустым, то весь входящий трафик будет заблокирован. Точно так же пустое поле egress блокирует весь исходящий трафик. Для более подробного описания разрешенных методов взаимодействия можно указывать списки портов и протоколов.

Поле policyTypes определяет, к какому типу сетевых политик относится объект NetworkPolicy. Если оно отсутствует, то Kubernetes будет учитывать только списки ingress и egress. Но опять же разница в том, что по умолчанию правила распространяются только на входящий трафик; при необходимости управлять исходящим трафиком вы должны явно указать egress в поле policyTypes и предоставить списки соответствующих правил.

Рассмотрим пример приложения, развернутого в едином пространстве имен и состоящего из трех уровней с метками tier: "web", tier: "db" и tier: "api". Если вы хотите корректно ограничить трафик тем или иным уровнем, то вам нужно создать такой манифест NetworkPolicy.

Правило, блокирующее весь трафик по умолчанию:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

Сетевая политика для веб-трафика:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: webaccess
spec:
  podSelector:
    matchLabels:
      tier: "web"
  policyTypes:
  - Ingress
  ingress:
  - {}
```

Сетевая политика для API:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-api-access
spec:
  podSelector:
    matchLabels:
      tier: "api"
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          tier: "web"
```

Сетевая политика для базы данных:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-db-access
spec:
  podSelector:
    matchLabels:
      tier: "db"
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          tier: "api"
```

Рекомендации по применению сетевых политик

Когда-то для защиты сетевого трафика в корпоративных системах использовалось физическое оборудование со сложными наборами сетевых правил. Теперь благодаря сетевым политикам в Kubernetes можно применять более программно-ориентированный подход, позволяющий разделять и контролировать трафик, принадлежащий приложениям кластера. Некоторые рекомендации актуальны независимо от того, какое сетевое дополнение вы используете.

- Начните с малого: сосредоточьтесь на трафике, предназначенном для pod. Применение правил для входящего и исходящего трафика может превратить трассировку сети в кошмар. Добившись подходящей маршрутизации запросов, вы можете уделить внимание правилам для исходящего трафика, чтобы обеспечить более жесткий контроль за чувствительными рабочими заданиями. Многие параметры по умолчанию относятся к входящему трафику, даже если поле `ingress` оставить пустым.
- Убедитесь, что используемое вами сетевое дополнение имеет собственный интерфейс для работы с API `NetworkPolicy` или поддерживает другие известные дополнения. Примерами таких дополнений могут служить Calico, Cilium, Kube-router, Romana и Weave Net.
- Если сетевые администраторы привыкли к тому, что весь трафик должен блокироваться по умолчанию, то создайте в каждом пространстве имен кластера, в котором будут выполняться рабочие задания, требующие защиты, сетевую политику наподобие той, что приведена ниже. Таким образом, даже в случае удаления других сетевых политик ни один pod не окажется случайно «незащищенным»:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

- При наличии pod, которые должны быть доступными из Интернета, используйте метки, чтобы явно применить к ним сетевые политики, разрешающие входящие запросы. Принимайте во внимание весь маршрут

следования трафика, если IP-адрес, с которого приходит пакет, принадлежит внутреннему серверу, балансировщику нагрузки, брандмауэру или другому сетевому устройству. Например, чтобы разрешить прием трафика из любых источников (в том числе и внешних) для pod с меткой `allow-internet=true`, сделайте следующее:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: internet-access
spec:
  podSelector:
    matchLabels:
      allow-internet: "true"
  policyTypes:
  - Ingress
  ingress:
  - {}
```

- Пытайтесь размещать все рабочие задания приложения в едином пространстве имен. Это облегчит создание правил, которые, как вы знаете, действуют в пределах одного пространства. Если вам нужно организовать взаимодействие пространств имен, то пытайтесь сделать это максимально явно. Для определения типов трафика можно использовать специальные метки:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: namespace-foo-2-namespace-bar
  namespace: bar
spec:
  podSelector:
    matchLabels:
      app: bar-app
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
       matchLabels:
         networking/namespace: foo
      podSelector:
       matchLabels:
         app: foo-app
```

- Создайте тестовое пространство имен с меньшим количеством запретительных политик (или вообще без таковых), чтобы иметь возможность подобрать подходящие методы управления трафиком.

Механизмы межсервисного взаимодействия

Вы можете легко представить кластер с сотнями сервисов, распределяющих нагрузки между тысячами конечных точек, которые взаимодействуют друг с другом, обращаются к внешним ресурсам и, возможно, принимают запросы снаружи. Администрирование, защита, отслеживание и трассировка всех соединений между сервисами могут быть непростой задачей, особенно учитывая динамическую природу конечных точек, принимающих и отправляющих трафик по всей системе. Концепция *межсервисного взаимодействия* (service mesh) позволяет контролировать соединения и безопасность этих сервисов с помощью отдельного уровня для работы с данными и управляющего уровня (data plane). Она не является уникальной для Kubernetes. На ее основе разработано множество механизмов с разными возможностями, однако все они, как правило, предоставляют следующие функции:

- балансировка трафика с помощью политик шейпинга, которые могут быть довольно подробными и распределяться по mesh-сети;
- обнаружение сервисов, входящих в mesh-сеть; могут находиться как в кластере (в том же или другом), так и во внешней системе;
- наблюдаемость трафика и сервисов, в том числе трассировка распределенных сервисов с помощью таких систем, как Jaeger или Zipkin, которые соответствуют стандартам OpenTracing;
- безопасность трафика в mesh-сети, основанная на взаимной аутентификации. В ряде случаев защищается трафик не только между pod или разными частями вашей системы, но и между вашей системой и внешними ресурсами;
- механизмы, обеспечивающие устойчивость, надежность и предотвращение сбоев, которые позволяют использовать такие паттерны проектирования, как «предохранители», повторные попытки, время ожидания и т. д.

Ключевое преимущество здесь в том, что интеграция всех этих замечательных возможностей не требует внесения существенных (или вообще никаких) изменений в приложения, участвующие в межсервисном взаимодействии. Как же получить все перечисленное без дополнительных усилий? Обычно для этого используются вспомогательные прокси-серверы. Большинство механизмов межсервисного взаимодействия, доступных на сегодняшний день, внедряют в каждый pod, входящий в mesh-сеть, прокси-сервер,

который является частью слоя для работы с данными. Благодаря этому политики и правила безопасности синхронизируются в mesh-сети с помощью управляющих компонентов. Это по-настоящему скрывает от контейнера с рабочими заданиями подробности сетевого взаимодействия и позволяет прокси-серверу взять на себя все нюансы распределенной сети. Приложение общается с прокси через localhost. Управляющий уровень и слой для работы с данными зачастую представляют собой разные, но взаимодополняющие технологии.

При упоминании межсервисного взаимодействия первыми на ум приходят такие проекты, как Istio от Google, Lyft и Envoy, который компания IBM использует в своем слое для работы с данными в качестве прокси-сервера вместе с проприетарными управляющими компонентами Mixer, Pilot, Galley и Citadel. Существуют и другие механизмы межсервисного взаимодействия с разной степенью функциональности — например, Linkerd2, использующий собственный прокси-сервер на уровне управления данными, написанный на Rust. Компания HashiCorp недавно добавила в Consul дополнительные возможности для взаимодействия сервисов, ориентированные на Kubernetes; которые, помимо коммерческой поддержки, позволяют выбрать между прокси-сервером, встроенным в Consul, и Envoy.

Тема механизмов межсервисного взаимодействия в Kubernetes весьма изменчива (и вызывает бурные дискуссии между техническими специалистами в социальных сетях), поэтому подробно описывать здесь каждую отдельную технологию нет смысла. Было бы упущением не упомянуть многообещающую инициативу вокруг Service Mesh Interface (SMI), возглавляемую компаниями Microsoft, Linkerd, HashiCorp, Solo.io, Kinvolk и Weaveworks. Задача этого проекта — создать стандартный интерфейс для основных возможностей, которые можно ожидать от любой mesh-сети. На текущий момент спецификация SMI охватывает политики управления трафиком, такие как идентификация, шифрование на транспортном уровне, телеметрия трафика, относящаяся к ключевым метрикам взаимодействия сервисов в mesh-сети, а также управление запросами с возможностью их смещения и взвешивания для разных сервисов. Разработчики этого проекта надеются сделать механизмы межсервисного взаимодействия более однородными, но так, чтобы поставщики могли расширять свои продукты и встраивать в них дополнительные возможности, позволяющие им выделиться на фоне конкурентов.

Рекомендации по применению механизмов межсервисного взаимодействия

Сообщество, сформировавшееся вокруг механизмов межсервисного взаимодействия, растет с каждым днем, и чем больше предприятий формулируют свои потребности, тем сильнее меняется эта экосистема. На сегодняшний день наиболее распространенные рекомендации основаны на общих потребностях, которые пытаются удовлетворить mesh-сети.

- Оцените степень важности ключевых возможностей, предлагаемых механизмами межсервисного взаимодействия, и определите, какие решения обладают самыми важными для вас свойствами и требуют меньше всего накладных расходов. Под накладными расходами здесь имеются в виду как человеческие усилия, так и инфраструктурные ресурсы. Если вам достаточно лишь TLS-соединения между определенными pod, то, вероятно, имеет смысл подобрать подходящее дополнение CNI.
- Нужны ли вам межсистемные mesh-сети (межоблачные и гибридные)? Не все механизмы межсервисного взаимодействия предоставляют такие возможности, и в любом случае это сложный процесс, который часто делает среду более хрупкой.
- Многие механизмы межсервисного взаимодействия являются открытыми проектами, которые разрабатываются сообществом. Как следствие, если команда, отвечающая за управление средой, не имеет опыта работы с этими технологиями, то решения с коммерческой поддержкой могут оказаться более подходящим вариантом. Некоторые компании начинают предоставлять услуги по коммерческой поддержке и администрированию mesh-сетей на основе Istio, что может быть полезно, поскольку мало кто станет отрицать, что Istio — довольно сложная в управлении система.

Резюме

Одна из самых важных функций Kubernetes, помимо управления приложениями, — возможность связывать разные участки вашей системы. В текущей главе мы подробно обсудили принципы работы Kubernetes, включая то, как pod получают свои IP-адреса через дополнения CNI, как эти адреса группируются в сервисы и как на основе ресурса Ingress (который, в свою очередь,

использует сервисы) можно реализовать новые приложения и средства маршрутизации прикладного уровня (Layer 7). Вы также узнали, каким образом можно ограничить трафик и защитить свою сеть с помощью сетевых политик и, наконец, какую роль технологии межсервисного взаимодействия играют в развитии коммуникаций между сервисами и их мониторинге. Помимо подготовки своего приложения для надежного развертывания, вам также необходимо позаботиться об организации сети в своей системе — это залог успешного применения Kubernetes. Вы должны понимать принципы Kubernetes по работе с сетью и то, как в данную схему вписывается ваше приложение.

ГЛАВА 10

Безопасность pod и контейнеров

Когда речь идет об обеспечении безопасности pod с помощью API Kubernetes, в вашем распоряжении есть два варианта: `PodSecurityPolicy` и `RuntimeClass`. В этой главе мы обсудим назначение каждого из этих API и дадим рекомендации по их использованию.

API PodSecurityPolicy

API PodSecurityPolicy все еще активно развивается. На момент выхода Kubernetes 1.15 он находился на этапе бета-тестирования. Актуальные сведения о его возможностях можно найти в официальной документации (oreil.ly/7UOWx).

Этот ресурс действует на уровне всего кластера и предоставляет вам единое место, где вы можете указывать и переопределять поля, относящиеся к безопасности pod (которые можно найти в их спецификациях). До появления ресурса `PodSecurityPolicy` администраторам и/или пользователям кластера приходилось определять параметры `SecurityContext` для отдельных рабочих заданий или использовать самописные контроллеры доступа для реализации тех или иных аспектов безопасности pod.

На первый взгляд объект `PodSecurityPolicy` может показаться крайне простым, но его корректная реализация требует на удивление много усилий, поэтому чаще всего его просто отключают или игнорируют. Тем не менее мы настоятельно рекомендуем вам основательно изучить `PodSecurityPolicy`, поскольку это одно из наиболее эффективных средств для уменьшения поверхности атак; с его помощью вы сможете ограничить рабочие задания, которым позволено выполняться в вашем кластере, и определить уровень их привилегий.

Включение PodSecurityPolicy

Соблюдение условий, определенных в объекте `PodSecurityPolicy`, требует включить как API для работы с ресурсами, так и контроллер доступа. То есть эти политики применяются на этапе проверки доступа в процессе обработки запроса. Больше информации о контроллере доступа можно узнать в главе 17.

Стоит отметить, что `PodSecurityPolicy` не имеет широкой поддержки в публичных облаках и средствах администрирования кластеров. Там, где этот объект доступен, он обычно предоставляется в качестве дополнительной возможности.

Будьте осторожны при включении объекта PodSecurityPolicy: если его заранее не настроить надлежащим образом, то он может заблокировать ваши рабочие задания.

Перед использованием `PodSecurityPolicy` вам следует выполнить два основных шага.

1. Убедиться в том, что у `PodSecurityPolicy` включен API (это должно быть сделано по умолчанию, если вы используете поддерживаемую версию Kubernetes).

   Используйте для этого команду `kubectl get psp`. Если вы не получили в ответ сообщение `the server doesn't have a resource type "PodSecurityPolicies"`, то, значит, все в порядке.

2. Включить контроллер доступа `PodSecurityPolicy` с помощью флага API-сервера `--enable-admission-plugins`.

Перед включением контроллера доступа PodSecurityPolicy в кластере с уже существующими рабочими заданиями следует заранее создать все необходимые политики, служебные учетные записи, роли и их привязки.

Мы также советуем указать дополнительный флаг `--use-service-account-credentials=true` для `kube-controller-manager`, чтобы включить служебные учетные записи для каждого отдельного контроллера, которым управляет данный сервис. Это обеспечит более гибкое применение политик даже внутри пространства имен `kube-system`. Узнать, был ли этот флаг установлен

ранее, можно с помощью следующей команды. Мы видим, что у каждого контроллера и в самом деле есть своя служебная учетная запись:

```
$ kubectl get serviceaccount -n kube-system | grep '.*-controller'
attachdetach-controller              1         6d13h
certificate-controller               1         6d13h
clusterrole-aggregation-controller   1         6d13h
cronjob-controller                   1         6d13h
daemon-set-controller                1         6d13h
deployment-controller                1         6d13h
disruption-controller                1         6d13h
endpoint-controller                  1         6d13h
expand-controller                    1         6d13h
job-controller                       1         6d13h
namespace-controller                 1         6d13h
node-controller                      1         6d13h
pv-protection-controller             1         6d13h
pvc-protection-controller            1         6d13h
replicaset-controller                1         6d13h
replication-controller               1         6d13h
resourcequota-controller             1         6d13h
service-account-controller           1         6d13h
service-controller                   1         6d13h
statefulset-controller               1         6d13h
ttl-controller                       1         6d13h
```

Ни в коем случае не забывайте, что при отсутствии политик PodSecurityPolicy любой доступ блокируется по умолчанию. Это значит, что pod не может быть создан, если его рабочее задание не имеет соответствующей политики.

Принцип работы PodSecurityPolicy

Чтобы разобраться в том, как `PodSecurityPolicy` помогает защитить ваши pod, рассмотрим полноценный пример. Так вы сможете запомнить порядок выполнения операций, начиная с создания политики и заканчивая ее применением.

Прежде чем продолжать, убедитесь в том, что в вашем кластере включена поддержка `PodSecurityPolicy`. Ее включение описано в предыдущем подразделе.

Перед включением PodSecurityPolicy в уже развернутом кластере ознакомьтесь с предупреждениями, приведенными в предыдущем подразделе. Соблюдайте осторожность.

Сначала посмотрим, как работает наш кластер при отсутствии каких-либо изменений или создания новых политик. Ниже приводится проверочное рабочее задание, которое просто запускает надежный контейнер `pause` в виде объекта `Deployment` (сохраните этот код в файле `pause-deployment.yaml` в своей локальной файловой системе; он нам еще пригодится):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pause-deployment
  namespace: default
  labels:
    app: pause
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pause
  template:
    metadata:
      labels:
        app: pause
    spec:
      containers:
      - name: pause
        image: k8s.gcr.io/pause
```

Указанная ниже команда позволяет подтвердить, что у вас есть `Deployment` с соответствующим объектом `ReplicaSet`, но *без* pod:

```
$ kubectl get deploy,rs,pods -l app=pause
NAME                                     READY  UP-TO-DATE  AVAILABLE  AGE
deployment.extensions/pause-delpoyment   0/1    0           0          41s

NAME                                               DESIRED  CURRENT  READY  AGE
replicaset.extensions/pause-delpoyment-67b77c4f69  1        0        0      41s
```

Чтобы узнать причину, можно проверить журнал событий `ReplicaSet`:

```
$ kubectl describe replicaset -l app=pause
Name:           pause-delpoyment-67b77c4f69
Namespace:      default
Selector:       app=pause,pod-template-hash=67b77c4f69
Labels:         app=pause
                pod-template-hash=67b77c4f69
Annotations:    deployment.kubernetes.io/desired-replicas: 1
                deployment.kubernetes.io/max-replicas: 2
                deployment.kubernetes.io/revision: 1
Controlled By:  Deployment/pause-delpoyment
```

```
Replicas:       0 current / 1 desired
Pods Status:    0 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=pause
           pod-template-hash=67b77c4f69
  Containers:
   pause:
    Image:        k8s.gcr.io/pause
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type             Status  Reason
  ----             ------  ------
  ReplicaFailure   True    FailedCreate
Events:
  Type     Reason        Age                  From                   Message
  ----     ------        ----                 ----                   -------
  Warning  FailedCreate  45s (x15 over 2m7s)  replicaset-controller  Error
creating:
pods "pause-delpoyment-67b77c4f69-" is forbidden: unable to validate against any
pod security policy: []
```

Описанное выше можно объяснить двумя причинами: либо у pod нет политик безопасности, либо служебная учетная запись не имеет доступа к `PodSecurityPolicy`. Вы также могли заметить, что все системные pod в пространстве имен `kube-system` до сих пор, вероятно, находятся в состоянии `RUNNING`. Это связано с тем, что данные запросы уже прошли этап контроля доступа. С ними произошло бы точно то же, что и с нашим пробным рабочим заданием, если бы возникло событие, которое перезапустило бы эти pod (при условии отсутствия ресурсов `PodSecurityPolicy`):

```
replicaset-controller  Error creating: pods "pause-delpoyment-67b77c4f69-" is
forbidden: unable to validate against any pod security policy: []
```

Удалим объект `Deployment` с пробным рабочим заданием:

```
$ kubectl delete deploy -l app=pause
deployment.extensions "pause-delpoyment" deleted
```

Теперь исправим это с помощью политик безопасности pod. Полный список их параметров можно найти в документации Kubernetes (oreil.ly/AsuVb). Там же приводятся примеры, на которых основаны приводимые здесь политики.

Назовем первую политику `privileged`; с ее помощью мы продемонстрируем, как выдавать рабочим заданиям повышенные привилегии. Следующие

ресурсы можно применить с использованием команды `kubectl create -f <имя_файла>`:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: privileged
spec:
  privileged: true
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - '*'
  volumes:
  - '*'
  hostNetwork: true
  hostPorts:
  - min: 0
    max: 65535
  hostIPC: true
  hostPID: true
  runAsUser:
    rule: 'RunAsAny'
  seLinux:
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'RunAsAny'
  fsGroup:'
    rule: 'RunAsAny'
```

Следующая политика определяет ограниченный доступ и подходит для многих рабочих заданий, кроме тех, которые отвечают за выполнение общекластерных сервисов, таких как `kube-proxy` в пространстве имен `kube-system`:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
spec:
  privileged: false
  allowPrivilegeEscalation: false
  requiredDropCapabilities:
    - ALL
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    - 'persistentVolumeClaim'
  hostNetwork: false
```

```
  hostIPC: false
  hostPID: false
  runAsUser:
    rule: 'RunAsAny'
  seLinux:
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'MustRunAs'
    ranges:
      - min: 1
        max: 65535
  fsGroup:
    rule: 'MustRunAs'
    ranges:
      - min: 1
        max: 65535
  readOnlyRootFilesystem: false
```

Убедиться в том, что политики были созданы, можно с помощью следующей команды:

```
$ kubectl get psp
NAME        PRIV   CAPS   SELINUX   RUNASUSER         FSGROUP    SUPGROUP
    READONLYROOTFS   VOLUMES
privileged  true   *      RunAsAny  RunAsAny          RunAsAny   RunAsAny
    false            *
restricted  false         RunAsAny  MustRunAsNonRoot  MustRunAs  MustRunAs
    false  configMap,emptyDir,projected,secret,downwardAPI,
persistentVolumeClaim
```

Итак, закончив с определением этих политик, мы должны разрешить их использование служебными учетными записями. Применим для этого систему управления доступом на основе ролей (Role-Based Access Control, RBAC).

Для начала создадим следующую роль `ClusterRole`, которая разрешает использовать ограниченную политику `PodSecurityPolicy`, созданную нами на предыдущем шаге:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp-restricted
rules:
- apiGroups:
  - extensions
  resources:
  - podsecuritypolicies
  resourceNames:
  - restricted
  verbs:
  - use
```

Создадим еще одну роль `ClusterRole`, которая разрешает использовать привилегированную политику `PodSecurityPolicy`, созданную на предыдущем шаге:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp-privileged
rules:
- apiGroups:
  - extensions
  resources:
  - podsecuritypolicies
  resourceNames:
  - privileged
  verbs:
  - use
```

Теперь необходимо создать соответствующую привязку `ClusterRoleBinding`, которая откроет группе `system:serviceaccounts` доступ к роли `psp-restricted`. Эта группа содержит все служебные учетные записи, принадлежащие контроллеру `kube-controller-manager`:

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp-restricted
subjects:
- kind: Group
  name: system:serviceaccounts
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: psp-restricted
  apiGroup: rbac.authorization.k8s.io
```

Создадим еще одно проверочное рабочее задание. Как видите, теперь у нас есть активный pod:

```
$ kubectl create -f pause-deployment.yaml
deployment.apps/pause-deployment created
$ kubectl get deploy,rs,pod
NAME                                   READY   UP-TO-DATE  AVAILABLE   AGE
deployment.extensions/pause-deployment  1/1     1           1           10s

NAME                                              DESIRED  CURRENT  READY  AGE
replicaset.extensions/pause-deployment-67b77c4f69  1        1        1      10s

NAME                                   READY   STATUS   RESTARTS   AGE
pod/pause-deployment-67b77c4f69-4gmdn   1/1     Running  0          9s
```

Обновим объект Deployment данного рабочего задания так, чтобы он нарушал политику ограничения. Для этого достаточно добавить поле privileged=true. Сохраните этот манифест в локальном файле pause-privileged-deployment.yaml и примените его с помощью команды kubectl apply -f <filename>:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pause-privileged-deployment
  namespace: default
  labels:
    app: pause
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pause
  template:
    metadata:
      labels:
        app: pause
    spec:
      containers:
      - name: pause
        image: k8s.gcr.io/pause
        securityContext:
          privileged: true
```

Вы опять можете видеть, что объекты Deployment и ReplicaSet действительно были созданы, чего нельзя сказать о pod. Причину этого можно найти в журнале событий ReplicaSet:

```
$ kubectl create -f pause-privileged-deployment.yaml
deployment.apps/pause-privileged-deployment created
$ kubectl get deploy,rs,pods -l app=pause
NAME                                                 READY  UP-TO-DATE
  AVAILABLE   AGE
deployment.extensions/pause-privileged-deployment  0/1    0
  0           37s

NAME                                                            DESIRED
  CURRENT   READY   AGE
replicaset.extensions/pause-privileged-deployment-6b7bcfb9b7  1
  0         0       37s
$ kubectl describe replicaset -l app=pause
Name:           pause-privileged-deployment-6b7bcfb9b7
Namespace:      default
Selector:       app=pause,pod-template-hash=6b7bcfb9b7
Labels:         app=pause
                pod-template-hash=6b7bcfb9b7
```

```
Annotations:    deployment.kubernetes.io/desired-replicas: 1
                deployment.kubernetes.io/max-replicas: 2
                deployment.kubernetes.io/revision: 1
Controlled By:  Deployment/pause-privileged-deployment
Replicas:       0 current / 1 desired
Pods Status:    0 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=pause
           pod-template-hash=6b7bcfb9b7
  Containers:
   pause:
    Image:        k8s.gcr.io/pause
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type             Status   Reason
  ----             ------   ------
  ReplicaFailure   True     FailedCreate
Events:
 Type     Reason        Age                   From                   Message
 ----     ------        ----                  ----                   -------
 Warning  FailedCreate  78s (x15 over 2m39s)  replicaset-controller  Error
creating: pods "pause-privileged-deployment-6b7bcfb9b7-" is forbidden: unable to
validate against any pod security policy: [spec.containers[0].securityContext.
privileged: Invalid value: true: Privileged containers are not allowed]
```

В этом листинге дается однозначный ответ: `Privileged containers are not allowed` (привилегированные контейнеры запрещены). Удалим объект `Deployment` с проверочным рабочим заданием.

```
$ kubectl delete deploy pause-privileged-deployment
deployment.extensions "pause-privileged-deployment" deleted
```

До сих пор мы имели дело только с привязками на уровне кластера. Разрешим проверочному рабочему заданию обращаться к привилегированной политике, используя служебную учетную запись.

Для начала нужно создать `serviceaccount` в пространстве имен по умолчанию:

```
$ kubectl create serviceaccount pause-privileged
serviceaccount/pause-privileged created
```

Привяжем `serviceaccount` к разрешительной роли `ClusterRole`. Сохраним этот манифест в локальном файле `pause-privileged-psp-permissive.yaml` и применим его с помощью команды `kubectl apply -f <`*`имя_файла`*`>`:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
```

```
metadata:
  name: pause-privileged-psp-permissive
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: psp-privileged
subjects:
- kind: ServiceAccount
  name: pause-privileged
  namespace: default
```

Наконец, обновим проверочное рабочее задание так, чтобы оно использовало служебную учетную запись `pause-privileged`. Затем выполним команду `kubectl apply`, чтобы применить задание к кластеру:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pause-privileged-deployment
  namespace: default
  labels:
    app: pause
spec:
  replicas: 1
  selector:
    matchLabels:
      app: pause
  template:
    metadata:
      labels:
        app: pause
    spec:
      containers:
      - name: pause
        image: k8s.gcr.io/pause
        securityContext:
          privileged: true
      serviceAccountName: pause-privileged
```

Как видите, теперь наш pod может использовать привилегированную политику:

```
$ kubectl create -f pause-privileged-deployment.yaml
deployment.apps/pause-privileged-deployment created
$ kubectl get deploy,rs,pod
NAME                                                 READY  UP-TO-DATE
  AVAILABLE   AGE
deployment.extensions/pause-privileged-deployment  1/1    1
  1           14s
```

```
NAME                                                          DESIRED
  CURRENT   READY   AGE
replicaset.extensions/pause-privileged-deployment-658dc5569f  1
  1         1       14s

NAME                                                READY  STATUS   RESTARTS  AGE
pod/pause-privileged-deployment-658dc5569f-nslnw  1/1    Running  0         14s
```

Чтобы узнать, какая политика PodSecurityPolicy была задействована, используйте следующую команду:

```
$ kubectl get pod -l app=pause -o yaml | grep psp
kubernetes.io/psp: privileged
```

Трудности при работе с PodSecurityPolicy

Теперь вы знаете, как конфигурировать и использовать `PodSecurityPolicy`, но стоит отметить, что применение этой политики в реальных средах связано с целым рядом проблем. В текущем подразделе мы опишем те из них, с которыми сталкивались сами.

Разумные политики по умолчанию

Благодаря политике `PodSecurityPolicy` администраторы и/или пользователи кластера могут быть уверены в том, что их рабочие задания отвечают определенному уровню безопасности; в этом ее реальная польза. Однако на практике многие рабочие задания по небрежности запускают от имени `root`; при этом используются тома `hostPath` или другие рискованные параметры, вынуждающие вас создавать дыры в безопасности ваших политик, без которых рабочие задания попросту не смогут выполняться.

От вас потребуется много усилий

Создание подходящих политик требует серьезного вложения времени и усилий, особенно если в Kubernetes уже присутствует много активных рабочих заданий, но при этом не включена поддержка `PodSecurityPolicy`.

Заинтересованы ли ваши разработчики в изучении PodSecurityPolicy

Захотят ли ваши разработчики изучать `PodSecurityPolicy`? Какие у них могут быть для этого стимулы? Без налаживания серьезной предварительной координации и автоматизации не стоит ожидать беспроблемного внедрения данной технологии — а это значит, что переход на нее, скорее всего, окажется провальным.

Громоздкий процесс отладки

Диагностика применения политики — непростая задача. Например, у вас может возникнуть желание разобраться в том, почему на ваше рабочее задание (не) распространяется та или иная политика. На сегодня для этого нет подходящих инструментов и средств ведения журнала.

Используете ли вы артефакты, которые находятся вне вашего контроля

Загружаете ли вы образы из Docker Hub или другого публичного репозитория? Есть вероятность того, что они будут тем или иным образом нарушать ваши политики и у вас не будет возможности их исправить. Еще один распространенный пример — Helm-чарты: поставляются ли они с подходящими политиками?

Рекомендации по использованию политики PodSecurityPolicy

У `PodSecurityPolicy` сложная конфигурация, которая может быть предрасположена к ошибкам. При реализации этой политики в своем кластере руководствуйтесь следующими рекомендациями.

- Все в конечном итоге сводится к RBAC. Нравится вам это или нет, политика `PodSecurityPolicy` определяется правилами доступа на основе ролей. Именно эта зависимость раскрывает все недостатки вашей ролевой модели. Сложно переоценить важность автоматизации создания и поддержки ваших ролей и политик `PodSecurityPolicy`. В частности, ключевой аспект использования этих политик — ограничение доступа к служебным учетным записям.
- Вы должны понимать, в каких пределах действует ваша политика. Очень важно иметь представление о том, каким образом она ложится на ваш кластер. Ваши политики могут распространяться на весь кластер, отдельные пространства имен или определенное рабочее задание. В кластере Kubernetes всегда есть системные pod, которым нужны повышенные привилегии безопасности, поэтому убедитесь в том, что ваши роли RBAC не дают использовать эти разрешающие политики нежелательным рабочим заданиям.
- Если вы хотите включить `PodSecurityPolicy` в существующем кластере, то используйте этот удобный и открытый инструмент (https://oreil.ly/2XLne), чтобы сгенерировать политики на основе имеющихся ресурсов. Это послужит отличным началом. А дальше вы уже сможете «оттачивать» свою конфигурацию.

PodSecurityPolicy: что дальше?

Как вы сами могли убедиться, `PodSecurityPolicy` представляет собой чрезвычайно эффективный API, который помогает сделать кластер безопасным, но при этом требует довольно серьезных усилий с вашей стороны. Тщательное планирование и прагматичный подход позволят вам успешно реализовать `PodSecurityPolicy` в любом кластере. По крайней мере, это будет хорошим подарком для вашего отдела безопасности.

Изоляция рабочих заданий и RuntimeClass

Среды выполнения контейнеров до сих пор в значительной степени считаются небезопасными с точки зрения изоляции. И пока нет никаких явных признаков того, что эта ситуация может измениться в будущем. Интерес к Kubernetes со стороны облачной индустрии привел к появлению разных контейнерных сред, степень изоляции которых варьируется. Часть из них основаны на известных технологических стеках, пользующихся доверием, а другие задействуют совершенно новый подход к этой проблеме. Проекты с открытым исходным кодом, такие как Kata Containers, gVisor и Firecracker, обещают повышенную изоляцию рабочих заданий. Они основаны либо на вложенной виртуализации (использование одной суперлегковесной виртуальной машины внутри другой), либо на фильтрации и предоставлении системных вызовов.

Появление контейнерных сред с разной изоляцией рабочих заданий позволяет пользователям применять в одном кластере разные решения с разными гарантиями. Например, вы можете разделить доверенные и сомнительные приложения между разными средами внутри одного кластера.

Для выбора контейнерных сред в Kubernetes был добавлен API `RuntimeClass`. Он используется на этапе конфигурации кластера для работы с одной из поддерживаемых сред выполнения контейнеров. Как пользователь Kubernetes, вы можете определять классы сред выполнения для своих рабочих заданий, используя `RuntimeClass` в спецификациях pod. Внутри Kubernetes создает обработчик `RuntimeHandler`, который передается в CRI (Container Runtime Interface) для дальнейшей реализации. Затем можно применять метки или ограничения в сочетании с селекторами `nodeSelector` или допусками, чтобы ваши pod были развернуты на узлах, поддерживающих нужную вам среду `RuntimeClass`. На рис. 10.1 показано, как утилита kubelet использует `RuntimeClass` при запуске pod.

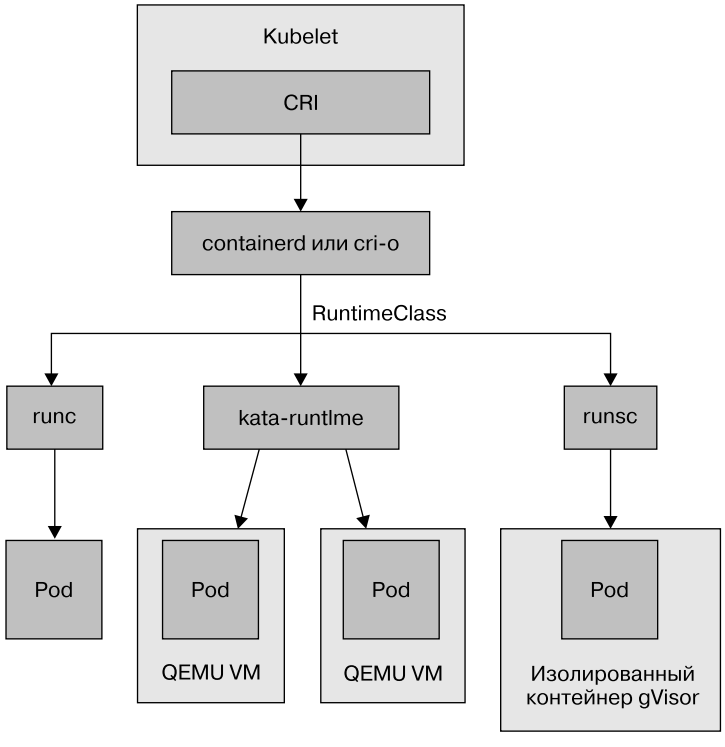


Рис. 10.1. Схема использования RuntimeClass

API RuntimeClass активно развивается. Последние новости о его возможностях можно найти в официальной документации (https://oreil.ly/N3KbO).

Использование RuntimeClass

Если администратор кластера подготовил разные среды RuntimeClass, то для их использования достаточно указать `runtimeClassName` в спецификации pod, например:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  runtimeClassName: firecracker
```

Реализации сред выполнения

Ниже перечислены некоторые открытые реализации сред выполнения контейнеров с разной степенью безопасности и изоляции на ваш выбор. Это лишь ориентировочный список, который ни в коем случае нельзя считать исчерпывающим.

- *CRI containerd* (oreil.ly/1wxU1) — обобщенный API для сред выполнения контейнеров с акцентом на простоту, надежность и переносимость.
- *cri-o* (cri-o.io) — легковесная реализация среды выполнения контейнеров на основе OCI (Open Container Initiative), предназначенная специально для Kubernetes.
- *Firecracker* (oreil.ly/on3Ge) — данная технология виртуализации работает поверх KVM (Kernel-based Virtual Machine) и позволяет очень быстро запускать крошечные ВМ в невиртуализированных средах, обеспечивая при этом безопасность и изоляцию традиционных виртуальных машин.
- *gVisor* (gvisor.dev) — изолированная среда выполнения на основе OCI. Запускает контейнеры с помощью нового ядра, работающего в пользовательском пространстве, что делает ее безопасной и изолированной, а также сокращает накладные расходы.
- *Kata Containers* (katacontainers.io) — сообщество, которое занимается разработкой безопасной среды выполнения контейнеров. Их решение предлагает безопасность и изоляцию на уровне ВМ за счет использования легковесных виртуальных машин, по своим характеристикам напоминающих контейнеры.

Изоляция рабочих заданий и рекомендации по использованию RuntimeClass

Ниже перечислены рекомендации, которые помогут вам избежать распространенных проблем с `RuntimeClass` и изоляцией рабочих заданий.

- Реализация разных изолированных сред выполнения контейнеров с помощью `RuntimeClass` усложнит ваше действующее окружение. Это значит, ваши рабочие задания, вероятно, нельзя будет переносить между разными средами, учитывая то, какого рода изоляцию они предоставляют. Иногда бывает сложно разобраться во всех поддерживаемых и неподдерживаемых возможностях разных сред выполнения, что делает их использование не-

удобным. Во избежание неразберихи мы рекомендуем по возможности задействовать отдельные кластеры для каждой среды выполнения.

- Изоляция рабочих заданий и безопасная мультиарендность — разные вещи. Даже если вы реализовали безопасное контейнерное окружение, это вовсе не означает, что кластер Kubernetes и API имеют аналогичную защиту. Вы должны принимать во внимание всю потенциальную поверхность атаки Kubernetes, от начала и до конца. Сам факт наличия изолированных рабочих заданий не гарантирует, что злоумышленник не сможет их модифицировать с помощью API кластера.
- Разные среды выполнения имеют разный инструментарий. Некоторые из ваших пользователей могут применять его для отладки и интроспекции. Наличие разных сред выполнения может означать, что у вас больше не будет возможности вывести список всех запущенных контейнеров с помощью команды `docker ps`. Это ведет к путанице и осложняет диагностику проблем.

Другие важные аспекты безопасности pod и контейнеров

Помимо `PodSecurityPolicy` и изоляции рабочих заданий, существуют другие инструменты, с помощью которых можно обеспечивать безопасность pod и контейнеров.

Контроллеры доступа

Если вы не хотите погружаться в изучение продвинутых аспектов `PodSecurityPolicy`, то в вашем распоряжении есть несколько менее функциональных, но вполне адекватных альтернатив. Для достижения похожего результата можно использовать контроллеры доступа, такие как `DenyExecOnPrivileged` и `DenyEscalatingExec`, в сочетании с веб-хуками; это позволит вам добавить в рабочие задания параметры `SecurityContext`. Подробней о контроле доступа см. в главе 17.

Средства обнаружения вторжений и аномалий

В данной главе мы рассмотрели политики безопасности и среды выполнения контейнеров, но что, если вам нужно анализировать и применять политики внутри самих сред? Для этих и других задач существуют открытые

инструменты, которые либо прослушивают и фильтруют системные вызовы Linux, либо применяют фильтр пакетов Berkeley (Berkeley Packet Filter, BPF). Одним из таких инструментов является Falco (https://falco.org/); это проект фонда Cloud Native Computing Foundation (CNCF), который устанавливается в виде Demonset и позволяет конфигурировать/применять политики во время выполнения. Но это лишь один из вариантов. Мы советуем вам исследовать инструменты, доступные в данной области, чтобы понять, какие из них подходят вам.

Резюме

В этой главе мы подробно обсудили API `PodSecurityPolicy` и `RuntimeClass`, позволяющие гибко конфигурировать уровень безопасности ваших рабочих заданий. Мы также рассмотрели некоторые инструменты с открытым исходным кодом, пригодные для мониторинга и применения политик внутри среды выполнения контейнеров. Вашему вниманию был предложен подробный обзор, на основе которого вы сможете принять обоснованное решение об уровне безопасности, необходимом вашим рабочим заданиям.

ГЛАВА 11

Политики и принципы управления кластером

Задумывались ли вы когда-нибудь о том, как сделать так, чтобы все контейнеры, запускаемые в кластере, загружались только из одного одобренного реестра? Возможно, у вас возникала необходимость надежно закрыть доступ к сервисам из Интернета. Это именно те проблемы, которые должны решаться с помощью политик и принципов управления кластером. Платформа Kubernetes становится все более зрелой, и круг предприятий, использующих ее, постоянно расширяется, поэтому возрастает актуальность данной области. Несмотря на ее относительную новизну и активное развитие, в текущей главе мы расскажем вам, как обеспечить совместимость вашего кластера с корпоративными политиками вашей организации.

Почему политики и принципы управления кластером имеют большое значение

Если вы работаете в жестко регулируемой области, такой как здравоохранение или финансы, или просто хотите иметь хоть какой-то контроль за тем, что выполняется в вашем кластере, то вам нужно каким-то образом реализовать задекларированные политики вашей организации. Составив список политик, вы должны определиться с тем, как воплотить их в жизнь и сделать так, чтобы ваши кластеры были с ними совместимы. Это может потребоваться для соблюдения нормативно-правовых норм или просто в целях следования общепринятым рекомендациям. Независимо от причины, вы должны позаботиться о том, чтобы реализация этих политик не сковывала ваших разработчиков и не ограничивала их автономность.

В чем отличие от других политик

Политика — ключевая концепция Kubernetes. Это могут быть сетевые политики или политики безопасности pod — но мы все понимаем, что они собой представляют и когда их следует использовать. Мы исходим из того, что содержимое спецификации ресурса реализуется в соответствии с определенными политиками. Объекты `NetworkPolicy` и `PodSecurityPolicy` применяются на этапе выполнения. Но кто отвечает за управление их спецификациями? Данная ответственность ложится на политики и принципы управления кластером. Вместо того чтобы работать на этапе выполнения, политика контролирует поля и значения в спецификациях самих ресурсов Kubernetes. И только те спецификации, которые соответствуют этим политикам, могут стать частью состояния кластера.

Облачно-ориентированная система политик

Чтобы понимать, какие ресурсы соответствуют политике, нам нужна достаточно гибкая система, отвечающая целому ряду требований. Агент Open Policy Agent (OPA) (https://www.openpolicyagent.org/) представляет собой открытую, гибкую и легковесную систему политик, которая набирает популярность в облачно-ориентированной экосистеме. Благодаря OPA мы имеем много разных реализаций инструментов управления Kubernetes. Вокруг одного из этих проектов, Gatekeeper (https://oreil.ly/RvKUw), сформировалось большое сообщество. В оставшейся части данной главы мы будем использовать Gatekeeper в качестве эталонного примера того, как реализовать политики и принципы управления кластером. Данный инструмент имеет альтернативы, но все они пытаются задействовать один и тот же подход, разрешая развертывать в кластере только те ресурсы Kubernetes, которые отвечают установленным политикам.

Введение в Gatekeeper

Gatekeeper — это открытый и гибкий веб-хук доступа для применения политик и принципов управления кластером. Он задействует систему ограничений OPA, чтобы вы могли описывать свои политики путем определения пользовательских ресурсов (custom resource definition, CRD). CRD обеспечивает тесную интеграцию с Kubernetes и разделяет процессы создания

и реализации политик. Шаблоны, которые при этом используются, называются *шаблонами ограничения* и могут разделяться и многократно применяться по всему кластеру. Gatekeeper позволяет проверять ресурсы и проводить аудит функциональности. Одна из замечательных особенностей данного инструмента — его переносимость; это значит, вы можете интегрировать его в любой кластер Kubernetes. Вы также можете попробовать импортировать в Gatekeeper политики OPA, если таковые имеются.

Gatekeeper все еще активно развивается и может заметно меняться. Последние новости о проекте ищите в его официальном репозитории (https://oreil.ly/Rk8dc).

Примеры политик

За всеми этими деталями очень важно не забыть о той проблеме, которую мы пытаемся решить. Для контекста рассмотрим отдельные политики, позволяющие решить наиболее распространенные вопросы соответствия различным требованиям:

- сервисы не должны быть публично доступны из Интернета;
- всем контейнерам следует загружаться из доверенных реестров;
- у всех контейнеров должны быть установлены лимиты на ресурсы;
- сетевым именам контроллеров Ingress не следует дублироваться;
- контроллеры Ingress должны использовать только HTTPS.

Терминология проекта Gatekeeper

В проекте Gatekeeper во многом применяются те же термины, которые используются и в OPA. Чтобы понять, как работает эта система, необходимо разбираться в ее терминологии. Gatekeeper задействует механизм ограничений OPA. Вот три новых термина, с которыми вы познакомитесь:

- ограничение (constraint);
- Rego;
- шаблон ограничений.

Ограничение

Ограничения применяются к определенным полям и значениям в спецификациях ресурсов Kubernetes. Это, в сущности, политики. Иными словами, вы фактически *запрещаете* то, что указано в определении ограничения. То есть ресурсы можно развертывать, если нет запрещающего ограничения. Это важно, поскольку вы блокируете только неподходящие вам поля и значения, разрешая все остальные. Данное архитектурное решение хорошо вписывается в концепцию спецификаций ресурсов, которые в Kubernetes постоянно меняются.

Rego

Rego — это язык запросов, встроенный в OPA. Запросы Rego представляют собой утверждения (assertions) о хранимых данных. Gatekeeper размещает их в шаблоне ограничений.

Шаблон ограничений

Это своеобразный шаблон политик. Он переносимый и подходит для многократного использования. Такие шаблоны состоят из типизированных параметров и запросов Rego с подставляемыми значениями.

Определение шаблона ограничений

Шаблоны ограничений являются пользовательскими определениями ресурсов (Custom Resource Definition, CRD; https://oreil.ly/LQSAH), которые предоставляют средства шаблонизации политик, чтобы их можно было разделять и использовать повторно. Кроме того, параметры политики могут проверяться. Рассмотрим шаблон ограничений в контексте предыдущих примеров. В следующем листинге показан шаблон, предоставляющий политику *«все контейнеры должны загружаться из доверенных реестров»*:

```
apiVersion: templates.gatekeeper.sh/v1alpha1
kind: ConstraintTemplate
metadata:
  name: k8sallowedrepos
spec:
  crd:
    spec:
      names:
        kind: K8sAllowedRepos
        listKind: K8sAllowedReposList
```

```
      plural: k8sallowedrepos
      singular: k8sallowedrepos
    validation:
      # структура поля `parameters`
      openAPIV3Schema:
        properties:
          repos:
            type: array
            items:
              type: string
targets:
  - target: admission.k8s.gatekeeper.sh
    rego: |
      package k8sallowedrepos

      deny[{"msg": msg}] {
        container := input.review.object.spec.containers[_]
        satisfied := [good | repo = input.constraint.spec.parameters.repos[_] ;
        good = startswith(container.image, repo)]
        not any(satisfied)
        msg := sprintf("container <%v> has an invalid image repo <%v>,
        allowed repos are %v", [container.name, container.image,
        input.constraint.spec.parameters.repos])
      }
```

Шаблон ограничений состоит из трех основных компонентов:

- *метаданные CRD, относящиеся к Kubernetes*, — самой важной частью является название. Позже мы будем на него ссылаться;
- *структура входящих параметров* — в поле `validation` описываются входящие параметры и их типы. В данном примере имеется лишь один параметр под названием `repo`, который представляет собой массив строк;
- *определение политики* — поле `target` содержит шаблонизированный запрос Rego (на языке, на котором описывают политики в OPA). Шаблон ограничений позволяет многократное использование таких запросов. Это значит, что разные компоненты могут разделять одну и ту же обобщенную политику. Если правило совпадает, то ограничение нарушается.

Определение ограничений

Чтобы использовать шаблон, определенный выше, нам нужно создать ресурс ограничения. Это делается в целях передачи в шаблон необходимых

параметров. Как видно ниже, в поле `kind` указан тип ресурсов `K8sAllowedRepos`, который привязывается к шаблону из предыдущего раздела:

```
apiVersion: constraints.gatekeeper.sh/v1alpha1
kind: K8sAllowedRepos
metadata:
  name: prod-repo-is-openpolicyagent
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    namespaces:
      - "production"
  parameters:
    repos:
      - "openpolicyagent"
```

Ограничение состоит из двух основных разделов:

- *метаданные Kubernetes* — обратите внимание на то, что тип этого ограничения, `K8sAllowedRepos`, совпадает с названием шаблона;
- *спецификация* — поле `match` определяет подходящую область действия политики. В этом примере нас интересуют только pod в пространстве имен `production`.

Параметры определяют назначение политики. Обратите внимание на то, что тип совпадает с тем, который был указан в описании структуры параметра в предыдущем разделе. В данном случае мы допускаем загрузку образов контейнеров только из тех репозиториев, названия которых начинаются с `openpolicyagent`.

Использование ограничений имеет следующие особенности.

- Ограничения объединяются с помощью логического И. Если одно и то же поле проверяется несколькими политиками и одна из них нарушается, то отклоняется весь запрос.
- Структура параметров проверяется, что позволяет заранее находить ошибки.
- Критерии выборки:
  - могут использовать метки-селекторы;
  - ограничивают только ресурсы определенных типов;
  - действуют лишь в определенных пространствах имен.

Репликация данных

Иногда может возникнуть необходимость сравнить текущий ресурс с другими ресурсами кластера, как в случае с политикой *«Сетевые имена контроллеров Ingress не должны пересекаться»*. Для проверки правила система OPA должна содержать в своем кэше все остальные ресурсы Ingress. Gatekeeper использует ресурс `config`, чтобы определить, какие данные должны кэшироваться в OPA. Ресурсы `config` также применяются для аудита, но об этом чуть позже.

Следующий ресурс `config` кэширует сервисы, pod и пространства имен версии v1:

```
apiVersion: config.gatekeeper.sh/v1alpha1
kind: Config
metadata:
name: config
  namespace: gatekeeper-system
spec:
  sync:
    syncOnly:
    - kind: Service
      version: v1
    - kind: Pod
      version: v1
    - kind: Namespace
      version: v1
```

Обратная связь

Gatekeeper в реальном времени сообщает пользователям кластера о ресурсах, нарушающих ту или иную политику. Возвращаясь к примерам из предыдущих разделов, мы разрешили загрузку контейнеров только из тех репозиториев, названия которых начинаются с `openpolicyagent`.

Попробуем создать следующий ресурс, который нарушает текущую политику:

```
apiVersion: v1
kind: Pod
metadata:
  name: opa
  namespace: production
spec:
  containers:
    - name: opa
      image: quay.io/opa:0.9.2
```

В результате вы получите сообщение о нарушении, которое было определено в шаблоне ограничений:

```
$ kubectl create -f bad_resources/opa_wrong_repo.yaml
Error from server (container <opa> has an invalid image repo <quay.io/opa:
0.9.2>, allowed repos are ["openpolicyagent"]): error when creating "bad_resour
ces/opa_wrong_repo.yaml": admission webhook "validation.gatekeeper.sh" denied
the request: container <opa> has an invalid image repo <quay.io/opa:0.9.2>,
allowed repos are ["openpolicyagent"]
```

Аудит

До сих пор мы говорили только об определении политики и применении ее в рамках процесса контроля доступа запросов. Но что делать с кластером, в котором уже развернуты ресурсы? Как узнать, какие из этих ресурсов соответствуют заданной политике? Именно здесь и нужен аудит. Если включить его, то Gatekeeper будет периодически проверять ресурсы на соответствие ограничениям, определенным вами. Это поможет выявить и исправить некорректную конфигурацию. Результаты аудита хранятся в поле `status` внутри ограничения, благодаря чему их легко найти с помощью утилиты `kubectl`. Ресурс, который проходит аудит, должен быть реплицирован. Подробнее об этом читайте в подразделе «Репликация данных» на с. 203.

Взглянем на ограничение под названием `prod-repo-is-openpolicyagent`, которое вы определили в предыдущем подразделе:

```
$ kubectl get k8sallowedrepos prod-repo-is-openpolicyagent -o yaml
apiVersion: constraints.gatekeeper.sh/v1alpha1
kind: K8sAllowedRepos
metadata:
  creationTimestamp: "2019-06-04T06:05:05Z"
  finalizers:
  - finalizers.gatekeeper.sh/constraint
  generation: 2820
  name: prod-repo-is-openpolicyagent
  resourceVersion: "4075433"
  selfLink: /apis/constraints.gatekeeper.sh/v1alpha1/k8sallowedrepos/prod-repois-
openpolicyagent
  uid: b291e054-868e-11e9-868d-000d3afdb27e
spec:
  match:
    kinds:
    - apiGroups:
      - ""
```

```
      kinds:
      - Pod
    namespaces:
    - production
  parameters:
    repos:
    - openpolicyagent
status:
  auditTimestamp: "2019-06-05T05:51:16Z"
  enforced: true
  violations:
  - kind: Pod
    message: container <nginx> has an invalid image repo <nginx>, allowed repos are
      ["openpolicyagent"]
    name: nginx
    namespace: production
```

Как видите, время, когда в последний раз проводился аудит, указано в поле `auditTimestamp`. А в поле `violations` можно видеть ресурсы, которые нарушают это ограничение.

Более тесное знакомство с Gatekeeper

В репозитории Gatekeeper есть фантастические примеры, в которых продемонстрированы все подробности создания политик, соответствующих нормативным требованиям банка. Мы настоятельно рекомендуем вам ознакомиться с ними, чтобы увидеть на практике, как работает Gatekeeper. Эти примеры можно найти в Git-репозитории по адресу https://oreil.ly/GcR3i.

Gatekeeper: что дальше?

Проект Gatekeeper продолжает расти, пытаясь решить другие проблемы в области политик и принципов управления. Речь идет о таких возможностях, как:

- мутация (модификация ресурсов в зависимости от политики; например, добавление определенных меток);
- внешние источники данных (интеграция с протоколом LDAP (Lightweight Directory Access Protocol — легкорасширяемый протокол доступа к каталогам) или Active Directory для поиска политик);
- авторизация (применение Gatekeeper в качестве модуля авторизации в Kubernetes);

- пробный запуск (дает пользователям возможность проверить политику, прежде чем активировать ее в кластере).

Если указанные проблемы вас заинтересовали и вы хотите помочь с их решением, то сообщество Gatekeeper всегда радо принять новых пользователей и разработчиков, желающих поучаствовать в развитии проекта. Больше об этом можно узнать в официальном репозитории на GitHub (https://oreil.ly/Rk8dc).

Рекомендации относительно политик и принципов управления

При реализации политик и принципов управления кластерами необходимо руководствоваться следующими рекомендациями.

- Если вы хотите применить политику к определенному полю pod, то должны выбрать для этого спецификацию подходящего ресурса. Возьмем, к примеру, объекты `Deployment`, управляющие контроллерами `ReplicaSet`, которые, в свою очередь, управляют pod. Мы могли бы применить политику на всех трех уровнях, но лучше выбрать ресурс, находящийся ближе всего к среде выполнения, — в данном случае это pod. Однако у этого решения есть некоторые последствия. При развертывании несовместимого pod вы не увидите подробное сообщение об ошибке, как в подразделе «Обратная связь» на с. 203. Дело в том, что несовместимый ресурс создает не пользователь, а объект `ReplicaSet`. То есть для обнаружения подобного ресурса пользователю нужно выполнить команду `kubectl describe` для текущего экземпляра `ReplicaSet`, связанного с развертыванием. И хотя эта процедура может показаться громоздкой, аналогичным образом себя ведут другие функции Kubernetes, например `PodSecurityPolicy`.
- Ограничения можно применять к ресурсам Kubernetes с учетом таких критериев, как тип (`kind`), пространство имен (`namespace`) и метка-селектор (`labelSelector`). Мы настоятельно рекомендуем делать область действия ограничений максимально узкой, охватывающей только те ресурсы, к которым вы хотите их применять. Это позволит сохранить согласованное поведение политик по мере расширения кластера. К тому же благодаря этому ресурсы, которые не нужно проверять, не будут передаваться в OPA, что повысит эффективность кластера.

- Синхронизация потенциально чувствительных данных, таких как объекты `Secret`, и применение к ним политик *не* рекомендуется. Вследствие того, что эти данные будут храниться в кэше OPA (если вы включили их репликацию) и передаваться в Gatekeeper, создается дополнительная поверхность для потенциального вектора атаки.
- Срабатывание хотя бы одного из ваших ограничений приводит к отклонению всего запроса. Эту функцию невозможно оформить в виде логического ИЛИ.

Резюме

В данной главе мы поговорили о важности политик и принципов управления кластером и прошлись по проекту, основанному на OPA (системе политик для облачно-ориентированной экосистемы), чтобы продемонстрировать, как эти концепции реализуются в Kubernetes. Теперь, если некто из отдела безопасности спросит вас: «Соответствуют ли наши кластеры политикам, которые мы определили?» — вы будете готовы и сможете дать уверенный ответ.

ГЛАВА 12

Управление несколькими кластерами

В текущей главе мы обсудим общепринятые рекомендации по управлению несколькими кластерами. Мы подробно поговорим о различиях между многокластерным администрированием и Federation, инструментами для управления множеством кластеров, и режимах работы с несколькими кластерами сразу.

Вы можете спросить, зачем вам это нужно, ведь платформа Kubernetes предназначена для объединения разных рабочих заданий в одном кластере. Это действительно так, но существуют специфические сценарии, такие как распределение рабочих заданий между регионами, минимизация последствий потенциальных проблем, соблюдение нормативно-правовых норм и выполнение специализированных приложений.

Мы обсудим все эти аспекты и исследуем инструменты и методики для управления несколькими кластерами в Kubernetes.

Зачем может понадобиться больше одного кластера

На момент внедрения Kubernetes у вас, скорее всего, будет больше одного кластера; возможно, вам придется иметь дело с отдельными средами для промышленных приложений, промежуточного тестирования, приемочного пользовательского тестирования (user acceptance testing, UAT) и разработки. Для поддержки мультиарендности в Kubernetes применяются пространства имен, которые разбивают кластер на более мелкие логические части. Пространства имен разделяют рабочие задания, позволяя определять роли RBAC, квоты, политики безопасности pod и сетевые политики. Это отличный механизм для разделения разных команд и проектов, но в ряде случаев его может недоставать. Ниже перечислены аспекты, на которые следует

обращать внимание при выборе между многокластерной и однокластерной архитектурами:

- ❑ минимизация последствий потенциальных проблем;
- ❑ соответствие нормативно-правовым нормам;
- ❑ безопасность;
- ❑ жесткая мультиарендность;
- ❑ региональные рабочие задания;
- ❑ специализированные рабочие задания.

При обдумывании архитектуры *минимизация последствий потенциальных проблем* должна быть в центре вашего внимания. Как показывает наш опыт, это одна из основных причин, почему пользователи выбирают многокластерные архитектуры. В микросервисных системах для минимизации ущерба применяются такие шаблоны проектирования, как предохранители, повторные попытки, партиционирование и ограничение трафика. Вы должны использовать эти методики и на уровне инфраструктуры, а наличие нескольких кластеров может предотвратить веерные сбои, вызываемые программными дефектами. Например, при наличии одного кластера, обслуживающего 500 приложений, в случае серьезных неполадок все они окажутся недоступными. Но если то же самое произойдет в системе с пятью кластерами, то пострадает только 20 % ваших приложений. Недостаток этого подхода в том, что теперь вам придется администрировать пять кластеров и степень консолидации ваших рабочих заданий будет не такой высокой. Дэн Вудс написал отличную статью (https://oreil.ly/YnGUD) о реальном каскадном сбое, произошедшем в промышленной среде Kubernetes. Это хорошая иллюстрация того, почему в крупных средах стоит использовать многокластерную архитектуру.

Соответствие нормативно-правовым нормам — это еще одна проблема, которую пытается решить многокластерная архитектура. Индустрия платежных карт, HIPAA (Health Insurance Portability and Accountability — закон о мобильности и подотчетности медицинского страхования) и другие рабочие задания требуют к себе особого отношения. И дело вовсе не в том, что Kubernetes не поддерживает какие-либо аспекты мультиарендности, просто такими приложениями легче управлять отдельно от заданий общего назначения. У них могут быть специфические требования относительно усиления безопасности, неразделяемых компонентов или выделенных

серверов. Вам будет намного проще поместить их в отдельную среду, чем адаптировать под них весь кластер.

Иногда в крупных кластерах Kubernetes сложно уследить за всеми аспектами *безопасности*. У каждой команды, которая использует ваш мультиарендный кластер, могут быть свои требования, и чем больше этих требований, тем сложнее их будет удовлетворить. Когда кластер становится слишком большим, у вас могут возникнуть проблемы даже с администрированием RBAC, сетевых политик и политик безопасности pod. Несущественное изменение сетевой политики может случайно создать дыру в безопасности других пользователей. При наличии же нескольких кластеров вы можете ограничить последствия некорректной конфигурации. Но если вы все же решите, что один крупный кластер Kubernetes удовлетворяет вашим требованиям, то убедитесь в том, что у вас налажен надежный рабочий процесс для изменения параметров безопасности; вы должны понимать, какие последствия может иметь обновление RBAC, сетевой политики или политики безопасности pod.

Платформа Kubernetes не поддерживает *жесткую мультиарендность*, поскольку все рабочие задания, запущенные в кластере, разделяют один и тот же API. Использование пространств имен позволяет организовать мягкую мультиарендность, но этого недостаточно, чтобы защититься от вредоносных приложений, развернутых в кластере. Многим пользователям жесткая мультиарендность не нужна, поскольку они доверяют своим рабочим заданиям; она, как правило, необходима облачным провайдерам и тем, кто использует ПО на основе SaaS или не доверяет сторонним приложениям, у которых нет адекватной системы управления пользователями.

При необходимости обслуживать региональный трафик ваша архитектура будет состоять из нескольких кластеров, размещенных в разных регионах. Многокластерный подход также требуется при глобальном развертывании приложений. Если некоторые из ваших рабочих заданий должны быть *регионально распределенными*, то это хороший повод использовать федеративные или множественные кластеры, и об этом мы поговорим чуть позже.

Специализированные рабочие задания, такие как высокопроизводительные вычисления (high-performance computing, HPC), машинное обучение (machine learning, ML) и грид-вычисления, тоже требуют применения многокластерной архитектуры. Им могут быть нужны особое аппаратное обеспечение, уникальные режимы производительности и специальные учетные записи. Как показывает наш опыт, эти требования не так сильно влияют на архитектурные решения, поскольку большинство из них (а именно,

специализированное оборудование и режимы производительности) можно удовлетворить с помощью пулов узлов Kubernetes. Если вам нужен очень большой кластер для HPC или машинного обучения, то подумайте о том, чтобы выделить для этих задач отдельные среды.

Использование множественных кластеров само по себе обеспечивает изоляцию рабочих заданий, но у этого подхода есть свои архитектурные проблемы, о которых нужно позаботиться с самого начала.

Проблемы многокластерной архитектуры

При выборе многокластерной архитектуры возникают определенные проблемы, которые могут сделать ее слишком сложной и тем самым воспрепятствовать ее внедрению. Отдельные области, в которых часто возникают такие проблемы, перечислены ниже:

- ❑ репликация данных;
- ❑ обнаружение сервисов;
- ❑ маршрутизация сети;
- ❑ эксплуатация;
- ❑ непрерывное развертывание.

Репликация и согласованность данных всегда вызывали трудности при развертывании приложений в разных географических регионах и кластерах. Вам следует определиться с тем, что и где будет выполняться, и разработать стратегию репликации. Большинство баз данных имеют для этого встроенные средства, но вы должны спроектировать свое приложение так, чтобы оно соответствовало выбранной вами стратегии. Базы данных типа NoSQL упрощают этот процесс, поскольку они могут распределять свое содержимое между несколькими серверами, но вам все равно нужно позаботиться о том, чтобы ваше рабочее задание умело справляться с отложенной согласованностью или хотя бы латентностью между разными географическими регионами. Некоторые облачные сервисы, такие как Google Cloud Spanner и Microsoft Azure CosmosDB, предоставляют БД, помогающие с распределением данных по нескольким географическим областям.

Каждый кластер Kubernetes развертывает собственный реестр для *обнаружения сервисов*, и если у вас есть несколько кластеров, то эти реестры не синхронизируются. Это затрудняет взаимную идентификацию и обнаружение

приложений. Такие инструменты, как Consul от HashiCorp, умеют прозрачно синхронизировать сервисы из разных кластеров — даже находящиеся за пределами Kubernetes. Существуют и другие средства, например Istio, Linkerd и Cillium, которые расширяют механизм обнаружения сервисов в многокластерной архитектуре.

Kubernetes существенно упрощает организацию сети внутри кластера, избегая иерархических структур и преобразования сетевых адресов (network address translation, NAT). Но все становится сложнее, если вам нужна маршрутизация входящего и исходящего трафика. Весь трафик направляется строго в один кластер, поскольку ресурс Ingress не поддерживает многокластерные топологии. Вам также нужно позаботиться об исходящем трафике между кластерами и его маршрутизации. В этом нет ничего сложного, если все ваше приложение находится в одном кластере, но в многокластерной архитектуре необходимо принимать во внимание латентность и дополнительные сетевые переходы сервисов, имеющих внешние зависимости. Тесно связанные между собой приложения имеет смысл развертывать в одном кластере; это позволит избавиться от латентности и дополнительных сложностей.

Один из крупнейших источников накладных расходов при работе с несколькими кластерами — *эксплуатация*. Вам придется поддерживать согласованность в большом количестве сред. При этом очень важно позаботиться о хорошем уровне автоматизации — так вы сможете уменьшить операционные затраты. В ходе автоматизации управления кластерами следует подумать об использовании дополнительных средств в целях развертывания и управления инфраструктурой. Например, Terraform от HashioCrp поможет вам поддерживать вашу армаду кластеров в согласованном состоянии.

Применение инструментов *IaC* (Infrastructure as Code — «инфраструктура как код»), таких как Terraform, позволяет сделать процесс развертывания кластеров воспроизводимым. Но, с другой стороны, у вас также должна быть возможность согласованного управления дополнительными возможностями кластера, такими как мониторинг, ведение журнала, управление трафиком, обеспечение безопасности и т. д. Безопасность — еще один важный аспект эксплуатации; вы должны иметь возможность управлять политиками безопасности, ролями RBAC и сетевыми политиками во всех кластерах сразу. Позже в данной главе мы подробно обсудим вопрос поддержания согласованности за счет автоматизации.

При использовании *непрерывного развертывания* (Continuous Delivery, CD) в многокластерной архитектуре API разных систем должны взаимодейство-

вать с одной и той же конечной точкой CD. Это чревато трудностями с доставкой приложений. Конечно, вы можете предусмотреть для этого сразу несколько рабочих процессов, но если их количество перевалит за сотню, то у вас возникнут большие проблемы. Учитывая это, вы должны искать разные пути выхода из данной ситуации. Позже мы рассмотрим некоторые решения, которые должны помочь вам.

Развертывание в многокластерной архитектуре

Одна из первых мер, которые необходимо принять при развертывании в многокластерной архитектуре, состоит в использовании средств IaC, таких как Terraform. Kubespray, kops или другие инструменты, предоставляемые облачными провайдерами, тоже подойдут — главное, чтобы с их помощью можно было организовать воспроизводимый процесс развертывания на основе системы контроля версий.

Автоматизация — залог успешного администрирования многокластерных сред. Вы должны пытаться автоматизировать все аспекты развертывания своего кластера, и ничего страшного, если это будет происходить постепенно.

Стоит упомянуть об одном интересном проекте под названием Cluster API (oreil.ly/edzIa), который все еще находится на стадии разработки. Он входит в состав Kubernetes и в будущем должен предоставлять декларативный API для создания, конфигурации и администрирования кластера. Cluster API расширяет возможности Kubernetes и позволяет создавать декларативную конфигурацию на уровне кластера, используя общий API. Таким образом, вы сможете с легкостью создавать инструменты для автоматизации кластеров. На сегодняшний день проект все еще находится в процессе разработки, поэтому обязательно отслеживайте его развитие.

Методики развертывания и администрирования

Kubernetes Operator — проект, реализующий концепцию *«инфраструктура как код»* (IaC). С помощью «операторов» можно абстрагировать развертывание приложений и сервисов в кластерах Kubernetes. Представьте, к примеру, что вы хотите внедрить Prometheus в качестве стандартного средства мониторинга ваших кластеров. Для этого вы должны подготовить для каждой среды и команды различные объекты (`Deployment`, `Service`, `Ingress` и т. д.). Кроме того, вам нужно поддерживать основные конфигурационные ресурсы Prometheus, такие как версии, хранилища, политики

удержания и реплики. Как вы можете себе представить, обслуживание подобного решения в масштабах большого количества кластеров и команд может быть довольно сложным.

Вместо того чтобы иметь дело с бесчисленными объектами и конфигурационными ресурсами, вы можете установить оператор `prometheus-operator`. Он расширит API Kubernetes и сделает доступными новые объекты, такие как `Prometheus`, `ServiceMonitor`, `PrometheusRule` и `AlertManager`; с их помощью вы сможете описать все детали развертывания Prometheus. Управлять этими и любыми другими объектами в API Kubernetes позволяет утилита `kubectl`.

Архитектура `prometheus-operator` показана на рис. 12.1.

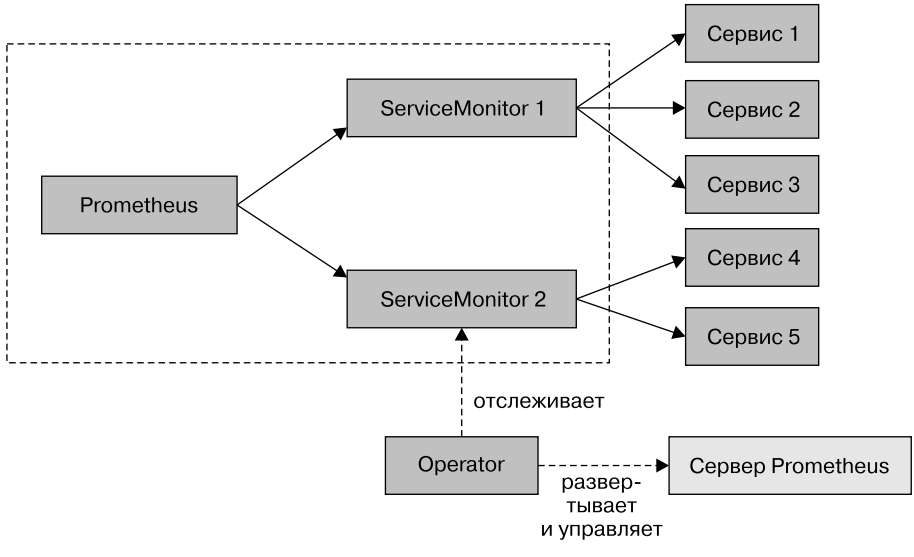


Рис. 12.1. Архитектура prometheus-operator

Использование *операторов* для автоматизации ключевых административных задач может улучшить общую управляемость кластера. Проект Operator был разработан командой CoreOS в 2016 году и изначально состоял из операторов для etcd и Prometheus. Он основан на двух концепциях:

- пользовательские определения ресурсов;
- пользовательские контроллеры.

Пользовательские определения ресурсов (custom resource definitions, CRD) — объекты, позволяющие описывать расширения для API Kubernetes.

Пользовательские контроллеры основаны на стандартных для Kubernetes концепциях: ресурсах и контроллерах. Они позволяют вам создавать собственную логику за счет отслеживания событий, которые генерируются объектами API Kubernetes, такими как `Deployment`, pod и ваши определения CRD. С помощью пользовательских контроллеров процесс создания CRD можно сделать декларативным. Контроллер `Deployment` в Kubernetes постоянно находится в цикле согласования, чтобы состояние объекта `Deployment` и его спецификация всегда были синхронизированы; ваши определения CRD имеют те же преимущества.

Kubernetes Operator позволяет автоматизировать управление средствами эксплуатации в многокластерной архитектуре. Рассмотрим в качестве примера оператор для Elasticsearch (https://oreil.ly/9WvJQ). В главе 3 мы использовали стек на основе Elasticsearch, Logstash и Kibana (ELK) в целях агрегации журнальных записей в нашем кластере. Оператор для Elasticsearch поддерживает следующие возможности:

- реплики для ведущих и клиентских узлов, а также узлов с хранилищами данных;
- зонирование для высокодоступных развертываний;
- установку размеров томов для ведущих узлов и узлов с хранилищами данных;
- изменение размера кластера;
- создание снимков для резервного копирования кластера Elasticsearch.

Как видите, этот оператор позволяет автоматизировать множество действий, которые приходится выполнять при администрировании Elasticsearch, в том числе создание снимков для резервного копирования и изменение размеров кластера. Прелесть данного подхода в том, что вы можете использовать уже знакомые вам объекты Kubernetes.

Подумайте о том, какие преимущества можно было бы извлечь из внедрения Kubernetes Operator в вашей среде и какие операторы вы могли бы написать самостоятельно для автоматизации рутинных административных задач.

Администрирование кластера с помощью методики GitOps

Методика *GitOps* была популяризирована работниками компании Weaveworks; их опыт использования Kubernetes в промышленных условиях воплотился в ее идеях и основных принципах. GitOps берет концепцию жизненного цикла разработки ПО и применяет ее к эксплуатации. Ваш репозиторий становится единым источником конфигурации, с которым синхронизируется кластер. Например, если вы обновите манифест `Deployment`, то внесенные изменения автоматически отразятся на состоянии вашей среды.

Этот метод позволяет упростить поддержку большого количества кластеров и избежать смещения конфигурации. Вы можете декларативно описать состояние кластеров для разных сред, и оно будет автоматически поддерживаться. Данный подход можно применять как к доставке приложений, так и к эксплуатации, но в текущей главе мы сосредоточимся на управлении кластерами и соответствующем инструментарии.

Weaveworks Flux был одним из первых инструментов, позволявших практиковать GitOps, и именно его мы будем использовать в оставшейся части этой главы. В облачно-ориентированной системе есть множество других, более новых инструментов; например, Argo CD от сотрудников Intuit тоже широко применяется для реализации концепции GitOps.

Рабочий процесс GitOps показан на рис. 12.2.

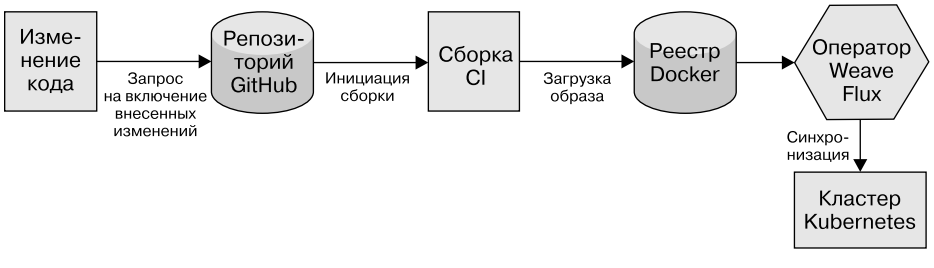


Рис. 12.2. Рабочий процесс GitOps

Установим Flux в вашем кластере и синхронизируем репозиторий с вашей средой:

```
git clone https://github.com/weaveworks/flux
cd flux
```

Теперь вам нужно отредактировать манифест `Deployment`, указав в нем ваш ответвленный репозиторий из главы 6. Откройте файл с манифестом:

```
vim deploy/flux-deployment.yaml
```

Подставьте в следующую строчку адрес своего Git-репозитория:

```
--git-url=git@github.com:weaveworks/flux-get-started
(например, --giturl=git@github.com:your_repo/kbp)
```

Теперь разверните Flux в своем кластере:

```
kubectl apply -f deploy
```

В ходе установки Flux создает SSH-ключ для аутентификации в Git-репозитории. Чтобы получить с его помощью доступ к своему репозиторию, воспользуйтесь утилитой командной строки `fluxctl`. Для начала ее нужно установить.

В MacOS:

```
brew install fluxctl
```

Для snap-пакетов в Linux:

```
snap install fluxctl
```

Последние пакеты для любых других систем можно найти по веб-адресу oreil.ly/4TAx5. Теперь выполните следующую команду:

```
fluxctl identity
```

Откройте GitHub и перейдите на страницу своего ответвленного репозитория. Откройте меню Setting ▸ Deploy keys (Настройки ▸ Ключи развертывания), щелкните на ссылке Add deploy key (Добавить ключ развертывания), введите название ключа, установите флажок Allow write access (Разрешить доступ на запись), вставьте открытый ключ Flux и нажмите кнопку Add Key (Добавить ключ). Подробнее о работе с ключами развертывания можно прочитать в документации GitHub.

Теперь, открыв журнал Flux, вы должны увидеть, что он синхронизирован с вашим репозиторием на GitHub:

```
kubectl -n default logs deployment/flux -f
```

Вслед за этим можно убедиться в создании pod для Elasticsearch, Prometheus, Redis и frontend-приложения:

```
kubectl get pods -w
```

Этот пример должен продемонстрировать, насколько просто синхронизировать состояние GitHub-репозитория с кластером Kubernetes. Это существенно облегчает управление большим количеством средств администрирования, поскольку разные кластеры могут синхронизироваться с одним и тем же репозиторием. При этом у вас больше не будет кластеров со смещенной конфигурацией.

Средства управления несколькими кластерами

Использование kubectl в многокластерной архитектуре может очень быстро привести к путанице, поскольку для управления разными кластерами необходимо устанавливать разные контексты. В таких случаях с самого начала следует установить два инструмента, *kubectx* и *kubens*, которые позволят легко переключаться между несколькими контекстами и пространствами имен.

В экосистеме Kubernetes имеется несколько полноценных решений для администрирования большого количества кластеров. Ниже приводится краткое описание некоторых из наиболее популярных проектов.

- *Rancher* занимается управлением несколькими кластерами Kubernetes с помощью централизованного пользовательского интерфейса. Этот инструмент поддерживает мониторинг, администрирование, резервное копирование и восстановление кластеров Kubernetes в облаках и локальных/удаленных вычислительных центрах. Он также позволяет управлять приложениями, развернутыми сразу в нескольких кластерах, и предоставляет инструментарий для эксплуатации.
- *KQueen* — это мультиарендный портал самообслуживания для выделения кластеров Kubernetes, основной упор в котором делается на аудит, прозрачность и безопасность в многокластерных архитектурах. KQueen — открытый проект, разработанный сотрудниками Mirantis.
- *Gardener* применяет другой подход к управлению множественными кластерами, предоставляя вашим конечным пользователям сервисы на основе стандартных компонентов Kubernetes. Этот инструмент поддерживает все основные облачные платформы, а в его разработке участвуют сотрудники SAP. Он в значительной степени ориентирован на пользователей, которые занимаются созданием SaaS на основе Kubernetes.

Kubernetes Federation

Система Federation v1 впервые появилась в Kubernetes 1.3 и уже успела устареть, а ей на смену пришла версия Federation v2. Federation v1 должна была помочь с распределением приложений между несколькими кластерами. Она использовала API платформы Kubernetes и сильно зависела от ее аннотаций, что создавало некоторые архитектурные проблемы. Тесная связь с основным API Kubernetes сделала систему Federation v1 довольно монолитной. Такая архитектура диктовалась не плохими решениями разработчиков, а скорее стандартными компонентами, доступными на тот момент. Появление Kubernetes CRD позволило по-новому взглянуть на то, как можно было бы спроектировать подобную систему.

Federation v2 (актуальное название *KubeFed*) поддерживается в Kubernetes 1.11 и выше и на сегодняшний день находится на стадии альфа-тестирования. Проект Federation v2 построен вокруг концепции CRD и пользовательских контроллеров, которые позволяют расширять Kubernetes с помощью новых API. Благодаря этому Federation может использовать в своих интерфейсах новые типы, не ограничиваясь одними лишь объектами `Deployment`, как версия v1.

Проект KubeFed направлен не столько на управление множественными кластерами, сколько на обеспечение высокодоступного многокластерного развертывания. Он предоставляет единую конечную точку, с помощью которой приложения могут развертываться сразу в нескольких кластерах Kubernetes. Например, имея кластеры, размещенные в разных облачных средах, вы можете объединить их в единый управляющий уровень и повысить устойчивость ваших приложений за счет многокластерного развертывания.

На момент написания этой книги поддерживаются следующие федеративные ресурсы:

- `Namespace`;
- `ConfigMap`;
- `Secret`;
- `Ingress`;
- `Service`;

- `Deployment`;
- `ReplicaSet`;
- `HorizontalPodAutoscaler`;
- `DaemonSet`;
- `Job`.

Чтобы понять, как все это работает, взгляните на архитектуру, представленную на рис. 12.3.

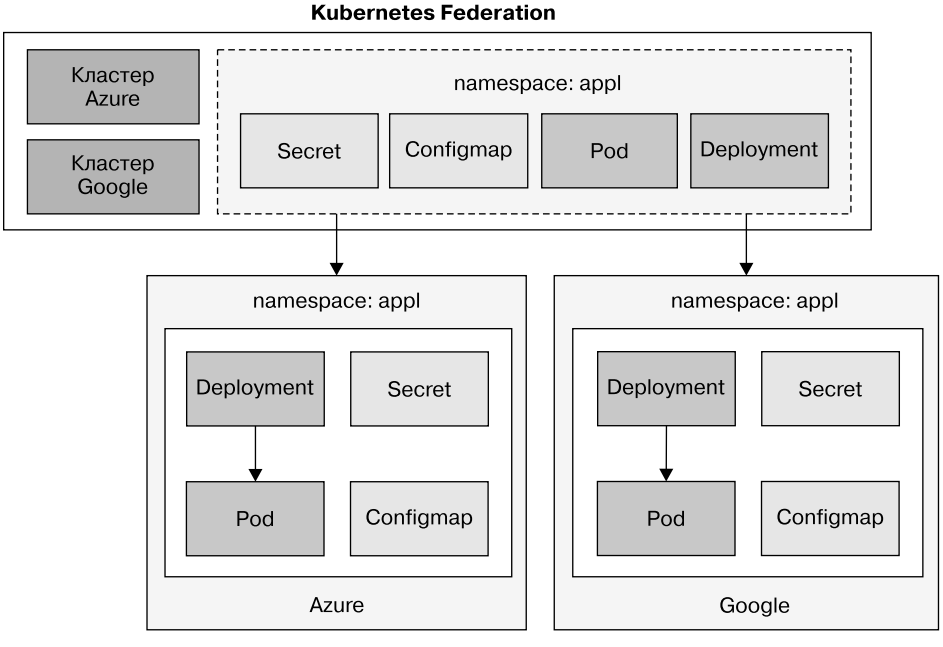


Рис. 12.3. Архитектура Kubernetes Federation

Необходимо понимать: использование Federation вовсе не означает, что все ресурсы просто копируются в каждый из кластеров. Например, по умолчанию в объектах `Deployment` и `ReplicaSet` указывается количество реплик, которые будут распределены между кластерами. Но вы можете изменить эту конфигурацию. С другой стороны, пространство имен создается в каждом кластере. Аналогичным образом работают объекты `Secret`, `ConfigMap` и `DaemonSet`. Ресурс Ingress ведет себя иначе; он создает глобальный многокластерный

объект с единой точкой входа в сервис. Из этого можно сделать вывод, что KubeFed подходит для межрегионального, межоблачного и глобального развертывания приложений в Kubernetes.

Ниже показан пример федеративного объекта `Deployment`:

```
apiVersion: types.kubefed.io/v1beta1
kind: FederatedDeployment
metadata:
  name: test-deployment
  namespace: test-namespace
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      replicas: 5
      selector:
        matchLabels:
          app: nginx
      template:
        metadata:
          labels:
            app: nginx
        spec:
          containers:
          - image: nginx
            name: nginx
  placement:
    clusters:
    - name: azure
    - name: google
```

Этот ресурс выполняет федеративное развертывание pod NGINX с пятью репликами, которые распределяются между кластерами в Azure и в Google.

Создание федеративных кластеров Kubernetes выходит за рамки данной книги. Больше об этом можно узнать в руководстве по использованию KubeFed (https://oreil.ly/tWmrY).

Проект KubeFed все еще находится на стадии альфа-тестирования. Вы можете следить за его развитием, но пока для обеспечения высокой доступности и выполнения многокластерного развертывания в Kubernetes лучше использовать уже доступные инструменты (или те, которые вы сами можете реализовать).

Рекомендации по эксплуатации сразу нескольких кластеров

При работе с несколькими кластерами Kubernetes руководствуйтесь следующими рекомендациями.

- Ограничьте последствия потенциальных проблем в ваших кластерах, чтобы минимизировать слияние веерных сбоев на ваши приложения.
- При необходимости соблюдать разные нормативы, такие как PCI, HIPPA или HiTrust, подумайте о вынесении подобных рабочих заданий в отдельные кластеры, чтобы не смешивать их с обычными приложениями.
- Если жесткая мультиарендность входит в число бизнес-требований, то рабочие задания следует развертывать в отдельном кластере.
- Если ваше приложение должно работать в разных регионах, то используйте глобальный балансировщик нагрузки для распределения трафика между кластерами.
- Специализированные рабочие задания, такие как HPC, можно размещать в отдельных кластерах, которые удовлетворяют их специфическим требованиям.
- Прежде чем развертывать рабочие задания, которые должны распределяться между несколькими региональными вычислительными центрами, необходимо позаботиться о стратегии репликации их данных. В межрегиональных кластерах как таковых нет ничего сложного, но репликация данных между регионами может вызвать затруднения. Поэтому вы должны иметь разумную стратегию для управления асинхронными и синхронными рабочими заданиями.
- Для работы с автоматизированными средствами администрирования используйте операторы из проекта Kubernetes operator, такие как `prometheus-operator` или Elasticsearch operator.
- При разработке многокластерной стратегии подумайте о том, как будут организованы обнаружение сервисов и сеть между кластерами. С созданием межкластерной сети могут помочь такие средства межсервисного взаимодействия, как Consul от HashiCorp.
- Убедитесь в том, что ваша стратегия CD подходит для множественного развертывания в разных регионах или кластерах.

- Рассмотрите возможность использования методики GitOps для работы с компонентами управления в многокластерной среде, чтобы обеспечить их согласованность во всех ваших кластерах. GitOps подходит не во всех ситуациях, но вы должны по крайней мере подумать о применении этого подхода для снижения операционных затрат в вашей многокластерной среде.

Резюме

В этой главе мы обсудили разные стратегии управления несколькими кластерами Kubernetes. Очень важно с самого начала подумать о потребностях вашей системы и о том, соответствуют ли эти потребности многокластерной топологии. В первую очередь следует рассмотреть необходимость в *жесткой* мультиарендности, так как она сама по себе требует применения многокластерной стратегии. Если она вам не нужна, то подумайте о нормативно-правовых требованиях и о том, хватит ли ваших операционных ресурсов для покрытия накладных расходов, связанных с многокластерной архитектурой. И наконец, располагая большим количеством мелких кластеров, позаботьтесь об автоматизации их развертывания и управления, чтобы уменьшить операционные затраты.

ГЛАВА 13

Интеграция внешних сервисов с Kubernetes

Во многих главах этой книги мы обсуждали сборку, развертывание и администрирование сервисов в Kubernetes. Но в реальности системы не находятся в вакууме и большинство сервисов, которые мы создаем, должны взаимодействовать с системами и сервисами, размещенными за пределами нашего кластера. Это может быть вызвано необходимостью взаимодействия со старой инфраструктурой, основанной на виртуальных машинах или физических серверах. Причиной также может послужить то, что нашим сервисам нужно обращаться к существующим базам данных или другим приложениям, работающим в рамках физической инфраструктуры в локальном вычислительном центре. И наконец, у вас может быть несколько разных кластеров Kubernetes с сервисами, которые нужно связать между собой. Как бы то ни было, способность создавать и разделять сервисы, выходящие за пределы вашего кластера, — важный аспект разработки реальных приложений.

Импорт сервисов в Kubernetes

Самый распространенный сценарий взаимодействия Kubernetes с внешними ресурсами — ситуация, когда сервис Kubernetes пользуется услугами, которые предоставляются за пределами его кластера. Это зачастую связано с тем, что Kubernetes используется в качестве среды разработки или интерфейса для старого ресурса, такого как локально размещенная база данных. Подобный подход часто имеет смысл в ходе инкрементальной разработки облачно-ориентированных сервисов. Поскольку слой БД содержит важные или незаменимые данные, его сложно перенести в облако, а тем более в контейнеры. Но в то же время создание поверх него современного слоя (такого как интерфейс GraphQL) имеет множество преимуществ и может стать основой для построения приложений нового поколения. Точно так же размещение данного слоя в Kubernetes является логичным шагом, поскольку его дина-

мичная разработка и надежное непрерывное развертывание обеспечивают высокую гибкость с минимальным риском. Конечно, чтобы этого достичь, вы должны сделать эту базу данных доступной из Kubernetes.

Первое, о чем следует позаботиться при налаживании доступа из Kubernetes к внешнему сервису, — корректная конфигурация сети. Особенности настройки сети сильно зависят от расположения базы данных и кластера Kubernetes, поэтому мы не станем в них углубляться. Но в целом облачные провайдеры позволяют развертывать кластер в пользовательской виртуальной сети, которую затем можно соединить с локальной.

Вслед за установлением соединения между pod кластера и локальным ресурсом необходимо сделать так, чтобы данный ресурс вел себя аналогично обычному сервису Kubernetes. В Kubernetes для обнаружения сервисов применяется DNS, поэтому в целях обеспечения работы с нашей внешней базой данных, как с любым другим компонентом кластера, мы должны сделать ее доступной на том же DNS-сервере.

Сервисы со стабильными IP-адресами без использования селекторов

Чтобы этого достичь, можно воспользоваться сервисом Kubernetes, у которого *нет селекторов*. Такому сервису не соответствуют никакие pod, вследствие чего для него не выполняется балансирование нагрузки. Вместо этого вы можете назначить ему определенный IP-адрес внешнего ресурса, который вам нужно добавить в кластер Kubernetes. Таким образом, когда pod попробует обратиться к вашей базе данных, встроенный в Kubernetes DNS-сервер передаст этот запрос IP-адресу внешнего сервиса. Вот пример сервиса для внешней БД без использования селекторов:

```
apiVersion: v1
kind: Service
metadata:
  name: my-external-database
spec:
  ports:
  - protocol: TCP
    port: 3306
    targetPort: 3306
```

Если сервис существует, то вы должны обновить его конечные точки так, чтобы они содержали IP-адрес базы данных, 24.1.2.3:

```
apiVersion: v1
kind: Endpoints
```

```
metadata:
  # Внимание! Это поле должно совпадать с именем сервиса.
  name: my-external-database
subsets:
  - addresses:
      - ip: 24.1.2.3
    ports:
      - port: 3306
```

На рис. 13.1 показано, как происходит интеграция внутри Kubernetes.

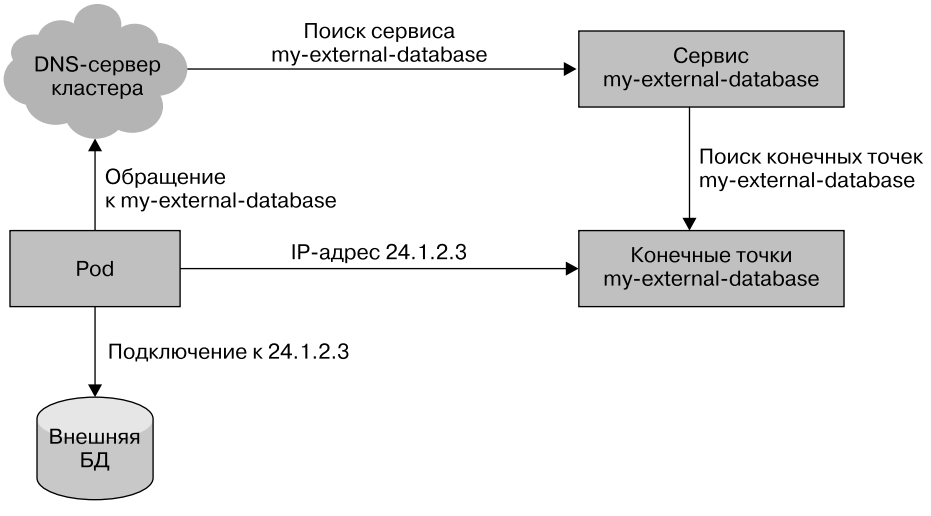


Рис. 13.1. Интеграция сервиса

Стабильные доменные имена сервисов на основе CNAME

В предыдущем примере предполагалось, что у внешнего ресурса, который вы пытаетесь интегрировать с кластером Kubernetes, есть стабильный IP-адрес. Это справедливо для многих физических, локально размещенных ресурсов, но зависит от топологии сети. А в облачных средах виртуальные машины (ВМ) имеют в основном динамические IP-адреса. Как вариант, у сервиса может быть несколько реплик, спрятанных за единым балансировщиком нагрузки на основе DNS. В этом случае у внешнего сервиса, который вы пытаетесь подключить к своему кластеру, нет стабильного IP-адреса, но зато есть стабильное доменное имя.

В таких ситуациях вы можете создать сервис Kubernetes на основе CNAME. Если вы не знакомы с DNS-записями, то CNAME (canonical name — «каноническое имя») сигнализирует о том, что одно доменное имя ссылается на другое. Например, если запись CNAME для foo.com содержит bar.com, то это говорит о том, что при открытии foo.com должен выполняться рекурсивный поиск bar.com, результатом которого должен быть корректный IP-адрес. Для определения записей CNAME на DNS-сервере можно использовать сервисы Kubernetes. Например, имея внешнюю базу данных с доменным именем database.myco.com, вы можете создать *сервис* на основе CNAME под названием `myco-database`. Он будет выглядеть примерно так:

```
kind: Service
apiVersion: v1
metadata:
  name: my-external-database
spec:
  type: ExternalName
  externalName: database.myco.com
```

В таком случае любой pod, который ищет `myco-database`, будет рекурсивным образом направлен к database.myco.com. Конечно, чтобы это работало, доменное имя вашего внешнего ресурса *тоже* должно быть прописано на DNS-сервере Kubernetes. При его глобальной доступности (например, оно предоставляется широко известным DNS-сервисом) все будет работать автоматически. Но если запись о доменном имени внешнего сервиса находится на локальном DNS-сервере компании (который, к примеру, обслуживает только внутренний трафик), то для направления запросов к нему, возможно, придется изменить стандартные параметры Kubernetes.

Чтобы DNS-сервер кластера мог найти внешний ресурс, вы должны откорректировать его конфигурацию. Для этого в манифесте `ConfigMap` следует указать конфигурационный файл DNS-сервера. На сегодня в большинстве кластеров используется CoreDNS. Для его настройки содержимое `corefile` необходимо записать в объект `ConfigMap` под названием `coredns`, который находится в пространстве имен `kube-system`. Если вы все еще используете сервер `kube-dns`, то он настраивается аналогичным образом, только с помощью другого объекта `ConfigMap`.

Записи CNAME — удобный механизм для привязывания внешних сервисов к стабильным доменным именам, которые можно найти внутри кластера. На первый взгляд связывание хорошо известного доменного имени с DNS-записью внутри кластера может показаться нелогичным, но это небольшое

усложнение обычно компенсируется тем, что все сервисы ведут себя одинаково. Кроме того, в Kubernetes сервис на основе CNAME, как и любой другой, существует в отдельном пространстве имен, поэтому в разных пространствах одно и то же доменное имя (скажем, `database`) может ссылаться на разные внешние сервисы (такие как `canary` или `production`).

Активный подход с применением контроллеров

В определенных редких случаях ни один из описанных выше методов не подходит для открытия доступа к внешним сервисам из Kubernetes. Обычно так происходит по причине того, что у внешнего сервиса нет ни стабильного доменного имени, ни единого статического IP-адреса. Это делает нашу задачу намного сложнее, однако она по-прежнему остается выполнимой.

Для начала разберемся с внутренним устройством сервисов Kubernetes. На самом деле сервисы в Kubernetes состоят из двух разных ресурсов: `Service`, с которым вы уже наверняка знакомы, и `Endpoints`, представляющего IP-адреса сервиса. В обычных условиях менеджер контроллеров Kubernetes формирует содержимое ресурса `Endpoints` на основе селекторов, указанных в манифесте `Service`. Но если у сервиса нет селекторов (как в примере со стабильным IP-адресом), то ресурс `Endpoints` останется пустым, поскольку Kubernetes не выберет ни один pod. В этой ситуации для наполнения ресурса `Endpoints` подходящими данными необходимо предоставить управляющий цикл. Вы должны динамически обращаться к своей инфраструктуре за IP-адресами внешних сервисов, которые хотите интегрировать в Kubernetes, и затем записывать эти адреса в `Endpoints`. Затем платформа Kubernetes сама корректно сконфигурирует DNS-сервер и `kube-proxy` для перенаправления трафика к вашему внешнему сервису. Полная картина того, как это работает на практике, представлена на рис. 13.2.

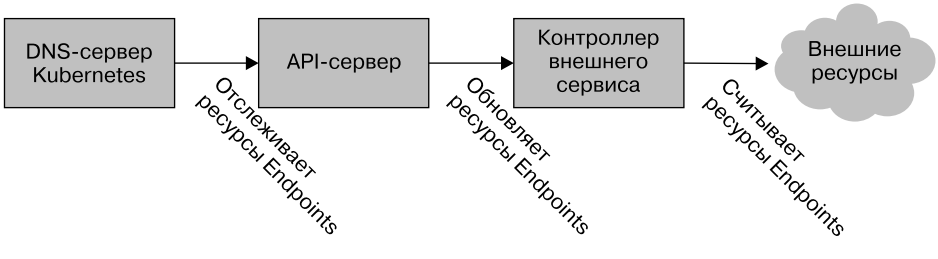


Рис. 13.2. Внешний сервис

Экспорт сервисов из Kubernetes

В предыдущем разделе мы обсудили импорт существующих сервисов в Kubernetes, однако иногда внутренние сервисы Kubernetes приходится экспортировать в существующие среды. Это может произойти, если у вас есть старое внутреннее приложение для работы с клиентами, которому нужно обращаться к какому-то новому API на основе облачной инфраструктуры. Возможно, вы разрабатываете новые API на основе микросервисов, которым ввиду внутренней политики или нормативно-правовых требований необходимо взаимодействовать с уже имеющимся традиционным брандмауэром веб-приложений (web application firewall, WAF). Какой бы ни была причина, во многих системах предоставление доступа к сервисам Kubernetes для других внутренних приложений — ключевое требование.

Основные трудности, возникающие в подобных ситуациях, связаны с тем, что во многих кластерах Kubernetes IP-адреса pod невозможно перенаправить наружу. Для маршрутизации внутри кластера используются такие инструменты, как flannel, которые обеспечивают взаимодействие pod друг с другом и с узлами, но их возможности обычно не распространяются на произвольные серверы в той же сети. Более того, если речь идет о сетевом соединении между облаком и локальным вычислительным центром, то VPN или пиринговые механизмы не всегда транслируют IP-адреса pod в локальную сеть. Следовательно, настройка маршрутизации между традиционными приложениями и pod — ключевой аспект экспорта сервисов, основанных на Kubernetes.

Экспорт сервисов с помощью внутреннего балансировщика нагрузки

Для экспорта из Kubernetes проще всего использовать стандартный объект `Service`. Если у вас есть некий опыт работы с Kubernetes, то вы уже, несомненно, знаете, как подключить облачный балансировщик нагрузки, чтобы направить внешний трафик к набору pod. Однако многие не знают, что большинство облаков предоставляют и *внутренний* балансировщик нагрузки, который тоже умеет связывать виртуальные IP-адреса с pod, только эти адреса берутся из локального диапазона (такого как 10.0.0.0/24), вследствие чего их можно маршрутизировать лишь внутри виртуальной сети. Для активации внутреннего балансировщика нагрузки в манифест сервиса нужно добавить аннотацию вашего облачного провайдера. Например, в Microsoft

Azure используется аннотация `service.beta.kubernetes.io/azure-load-balancerinternal: "true"`, а в Amazon Web Services (AWS) — `service.beta.kubernetes.io/aws-load-balancer-internal: 0.0.0.0/0`. Аннотацию необходимо указать в разделе `metadata` ресурса `Service`, как показано ниже:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  annotations:
    # При необходимости замените это в других средах
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
...
```

При экспорте сервиса через внутренний балансировщик нагрузки вы получаете стабильный IP-адрес, который поддается маршрутизации и доступен в виртуальной сети за пределами кластера. Этот IP-адрес затем можно использовать либо напрямую, либо с помощью внутреннего DNS-сервера, чтобы сделать экспортируемый сервис доступным для обнаружения.

Экспорт сервисов типа NodePort

К сожалению, при локальном размещении облачные провайдеры не могут предоставить внутренние балансировщики нагрузки. В этом контексте использование сервисов типа `NodePort` часто оказывается хорошим решением. Такие сервисы экспортируют на каждом узле кластера прослушивающий pod, который перенаправляет трафик из IP-адреса и выбранного порта узла к заданному вами объекту `Service`. Это показано на рис. 13.3.

Вот пример YAML-файла для сервиса типа `NodePort`:

```
apiVersion: v1
kind: Service
metadata:
  name: my-node-port-service
spec:
  type: NodePort
...
```

Вслед за созданием сервиса с типом `NodePort` Kubernetes автоматически выбирает для него порт. Для того чтобы узнать его номер, можно проверить поле `spec.ports[*].nodePort`. Порт можно указать и самостоятельно, во время создания сервиса, однако он должен находиться в диапазоне, который определен в конфигурации кластера. По умолчанию это номера от 30 000 до 32 767.

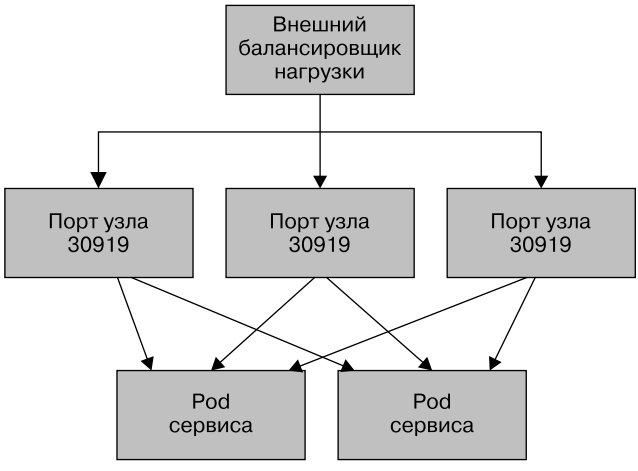


Рис. 13.3. Сервис типа NodePort

Участие Kubernetes в данном процессе завершается, когда сервис становится доступным на этом порте. Чтобы экспортировать сервис в существующее приложение за пределами кластера, вам (или администратору вашей сети) необходимо сделать так, чтобы это можно было обнаружить. В зависимости от конфигурации вашего приложения вы можете предоставить ему список пар `${`*`узел`*`}:${`*`порт`*`}`, и оно само займется балансировкой нагрузки на клиентской стороне. Возможно, вам придется настроить физический или виртуальный балансировщик нагрузки в пределах вашей сети, чтобы направлять трафик из виртуального IP-адреса к данному списку серверов `${`*`узел`*`}:${`*`порт`*`}`. Детали этой конфигурации будут варьироваться в зависимости от вашей среды.

Интеграция внешних серверов в Kubernetes

Если ни одно из предложенных выше решений вам не подходит (возможно, вам нужна более тесная интеграция для динамического обнаружения сервисов), то у вас остается последний вариант: интегрировать сервер (-ы) с внешним приложением непосредственно в механизмы кластера Kubernetes для обнаружения сервисов и управления сетью. Данный подход куда более радикальный и сложный по сравнению с рассматривавшимися ранее, и его следует применять только в случаях, когда это действительно необходимо (что должно быть нечасто). В некоторых управляемых средах Kubernetes он и вовсе невозможен.

При интеграции внешнего сервера в кластер необходимо убедиться в том, что маршрутизация pod и обнаружение сервисов на основе DNS работают корректно. На самом деле для этого проще всего запустить kubelet на компьютере, который вы хотите присоединить, и при этом отключить планировщик в кластере. Подключение узла kubelet выходит за рамки нашей книги, однако на данную тему есть большое количество материала. Сразу после присоединения узла его необходимо исключить из процесса планирования с помощью команды `kubectl cordon ...`, чтобы на нем не развертывались другие рабочие задания. DaemonSet по-прежнему сможет размещать pod на этом узле, благодаря чему на нем будут установлены KubeProxy и средства сетевой маршрутизации; это позволит любому приложению, которое выполняется на данном сервере, обнаруживать сервисы Kubernetes.

Описанный подход довольно агрессивен, поскольку требует установки на узле Docker или какой-то другой среды выполнения контейнеров. В связи с этим его нельзя применять во многих средах. У него есть более легковесная, но в то же время сложная альтернатива: запустить на сервере `kube-proxy` в виде обычного процесса и откорректировать его DNS-конфигурацию. Если у вас есть возможность должным образом настроить маршрутизацию pod, то `kube-proxy` сконфигурирует сеть узла так, чтобы виртуальные IP-адреса сервисов Kubernetes перенаправляли трафик к их pod. Если указать в сетевых настройках внешнего узла DNS-сервер кластера Kubernetes, то это фактически активирует обнаружение сервисов на данном узле.

Оба подхода сложны и требуют определенных навыков, поэтому, прежде чем их выбирать, нужно все тщательно взвесить. Если вы задумываетесь о таком уровне интеграции механизма обнаружения сервисов, то спросите себя, не легче ли будет взять сервис, который вы хотите подключить, и сделать его частью кластера.

Разделение сервисов между кластерами Kubernetes

В предыдущих разделах описывалось то, как подключать приложения Kubernetes к внешним сервисам и наоборот, но существует еще один распространенный сценарий: подключение сервисов к *разным* кластерам Kubernetes. Это может потребоваться на случай отказа отдельных региональных класте-

ров или для объединения сервисов, которыми занимаются разные команды. Чтобы наладить такое взаимодействие, нам потребуется сочетание методик, описанных в предыдущих разделах.

Прежде всего, чтобы обеспечить маршрутизацию сетевого трафика, вы должны сделать ваш сервис доступным в рамках первого кластера Kubernetes. Предположим, вы используете облачную среду с поддержкой внутренних балансировщиков нагрузки и ваш балансировщик имеет виртуальный IP-адрес 10.1.10.1. Чтобы сделать возможным обнаружение сервисов, вам нужно интегрировать данный IP-адрес во второй кластер. Здесь используется тот же подход, что и при импорте внешних приложений в Kubernetes (первый раздел). Вам нужно создать сервис без селекторов и назначить ему IP-адрес 10.1.10.1. Благодаря выполнению этих двух шагов вы получите интегрированное обнаружение и соединение между сервисами в рамках ваших двух кластеров.

Данный процесс требует существенного ручного вмешательства, но это вполне приемлемо для небольшого статического набора сервисов. Если же вам нужна более тесная или автоматическая интеграция сервисов между кластерами, то имеет смысл написать программу (демон), которая будет работать в обоих кластерах и заниматься интеграцией. Эта программа будет отслеживать в первом кластере все объекты `Service` с определенной аннотацией, такой как `myco.com/exported-service`, и импортировать их во второй кластер с помощью сервисов без селекторов. Та же программа будет собирать и удалять любые сервисы, экспортированные во второй кластер, но больше не присутствующие в первом. В результате все ваши региональные кластеры, в которых установлена эта программа, будут иметь динамическую взаимную связь.

Сторонние инструменты

До сих пор в текущей главе описывались разные методы импорта, экспорта и соединения сервисов, выходящих за пределы кластеров Kubernetes, и некоторых внешних ресурсов. Если у вас уже есть опыт работы с технологиями межсервисного взаимодействия, то эти концепции могут показаться вам довольно знакомыми. Действительно, существует множество сторонних инструментов и проектов, позволяющих связывать сервисы как с Kubernetes, так и с произвольными приложениями и серверами. В целом эти инструменты

имеют богатые возможности, но в то же время их намного сложнее использовать по сравнению с методиками, описанными ранее. Тем не менее если вам все чаще приходится заниматься налаживанием сетевых связей, то вы должны исследовать область mesh-сетей, активно развивающуюся сейчас. Почти все сторонние технологии имеют открытый исходный код, но часть из них также предлагают коммерческую поддержку, которая может снизить операционные затраты, связанные с обслуживанием дополнительной инфраструктуры.

Рекомендации по соединению кластеров и внешних сервисов

- Установите сетевое соединение между кластером и локально размещенной системой. Сетевая конфигурация может зависеть от настроек сайтов, облаков и кластеров; главное, убедитесь в том, что ваши pod могут взаимодействовать с локальными серверами и наоборот.
- Для доступа к приложениям за пределами кластера можно использовать сервисы без селекторов, напрямую указывая IP-адрес сервера (например, базы данных), с которым вы хотите взаимодействовать. Если у вас нет статических IP-адресов, то можете использовать вместо них записи CNAME с перенаправлением к доменному имени. Если же у вас нет и доменных имен, то вам, вероятно, придется написать динамический оператор, который будет время от времени синхронизировать IP-адреса внешнего сервиса с конечными точками объекта `Service` в Kubernetes.
- Для экспорта сервисов из Kubernetes используйте внутренние балансировщики нагрузки или объекты `Service` типа NodePort. Внутренние балансировщики обычно проще использовать в публичных облачных средах, в которых их можно привязать к самим сервисам Kubernetes. Если же этот способ недоступен, то объекты `Service` типа NodePort могут экспортировать сервисы со всех узлов кластера.
- Чтобы наладить сетевое соединение между кластерами, эти два подхода можно комбинировать: сделайте сервис доступным снаружи, а затем обращайтесь к нему из другого кластера Kubernetes с помощью объектов `Service` без селекторов.

Резюме

На практике не все приложения являются облачно-ориентированными. Построение реальных приложений часто подразумевает взаимодействие с уже существующими системами. В данной главе были описаны процессы интеграции Kubernetes со старыми приложениями и разделения сервисов между несколькими отдельными кластерами Kubernetes. Создание систем с нуля — роскошь, доступная не всем, поэтому интеграция со старыми сервисами всегда будет частью облачно-ориентированной разработки. В ее реализации вам помогут методики, описанные в текущей главе.

ГЛАВА 14

Машинное обучение и Kubernetes

Эпоха микросервисов, распределенных систем и облаков создала идеальные условия для демократизации моделей и средств машинного обучения (machine learning, ML). Масштабируемая инфраструктура превратилась в продукт, а инструментарий вокруг экосистемы машинного обучения становится все более зрелым. И так получилось, что платформа Kubernetes, наряду с другими системами, приобрела большую популярность среди специалистов, занимающихся анализом данных, и в сообществе открытого программного обеспечения, превратившись в идеальную среду для организации рабочего процесса и жизненного цикла ML. В этой главе мы покажем, за счет чего Kubernetes так хорошо подходит для машинного обучения, и дадим советы администраторам кластеров и специалистам по анализу данных относительно того, как в Kubernetes максимально эффективно налаживать рабочие задания для ML. В частности, вместо традиционного машинного обучения мы сосредоточимся на так называемом глубоком обучении, которое быстро стало инновационной областью в контексте таких платформ, как Kubernetes.

Почему Kubernetes отлично подходит для машинного обучения

Платформа Kubernetes быстро стала эпицентром бурных инноваций в глубоком обучении. Сочетание инструментария и библиотек, таких как TensorFlow, сделало эту технологию более доступной для широкого круга специалистов по анализу данных. Но за счет чего Kubernetes оказывается отличным выбором для выполнения процессов глубокого обучения? Перечислим основные свойства этой платформы.

- *Широкое распространение*. Платформа Kubernetes используется повсеместно. Ее поддерживают все крупные облачные провайдеры, у нее есть дистрибутивы для закрытых облаков и локальной инфраструкту-

ры. Задействуя Kubernetes в качестве основы для своей экосистемы, вы сможете выполнять процессы глубокого обучения практически везде.

- *Масштабируемость*. Для эффективного моделирования процессам глубокого обучения обычно нужен доступ к большому объему вычислительных ресурсов. Kubernetes имеет встроенные средства автоматического масштабирования, позволяющие специалистам по анализу данных легко подбирать тот масштаб, который требуется для совершенствования их моделей.
- *Расширяемость*. Для эффективного обучения модели ML, как правило, требуется доступ к специализированному оборудованию. Kubernetes позволяет инженерам быстро и легко выделять новое оборудование без изменения исходного кода платформы. Кроме того, в API Kubernetes можно без труда интегрировать пользовательские ресурсы и контроллеры для поддержки специализированных рабочих заданий, таких как оптимизация гиперпараметров.
- *Самообслуживание*. Специалисты по анализу данных могут самостоятельно использовать Kubernetes для выполнения рабочих заданий ML, не имея никаких специальных знаний об этой платформе.
- *Переносимость*. Модели машинного обучения можно выполнять где угодно, если ваш инструментарий основан на API Kubernetes. Благодаря этому вы можете переносить свои рабочие задания между разными провайдерами.

Рабочий процесс машинного обучения

Для основательного понимания того, что необходимо для глубокого обучения, нужно сначала осознать, как выглядит весь рабочий процесс ML. Его упрощенную версию можно видеть на рис. 14.1.

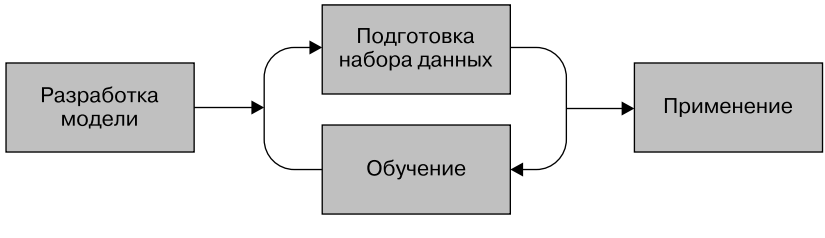


Рис. 14.1. Процесс разработки модели ML

Как видите, процесс разработки модели машинного обучения состоит из следующих этапов.

- *Подготовка набора данных* — этап включает в себя хранение, индексацию, каталогизацию и создание метаинформации на основе набора данных, используемого в обучении модели. В нашей книге мы сосредоточимся только на хранении. Наборы данных, с помощью которых совершенствуется модель, могут быть разного размера, от сотен мегабайт до сотен терабайт. Вы должны подобрать хранилище, удовлетворяющее вашим требованиям. Обычно для этого используются блочные и объектные хранилища, которые должны быть доступны либо через стандартный слой абстракции Kubernetes, либо напрямую через API.
- *Разработка алгоритма машинного обучения* — этап, на котором специалист по анализу данных пишет и публикует свои алгоритмы ML, возможно, в сотрудничестве с другими людьми. В Kubernetes легко устанавливаются такие открытые инструменты, как JupyterHub, поскольку они обычно ведут себя аналогично любому другому рабочему заданию.
- *Обучение* — процесс, в ходе которого модель берет набор данных и учится выполнять те задачи, для которых была создана. Итоговым результатом обычно является срез состояния обученной модели. Здесь применяются сразу все возможности Kubernetes: планирование, доступ к специализированному оборудованию, управление томами с наборами данных, масштабирование и организация сети. Подробности этого процесса рассматриваются в следующем разделе.
- *Применение* — на данном этапе обученная модель становится доступной для обработки клиентских запросов; она берет данные, предоставленные клиентом, и генерирует предсказание. Например, если у вас есть модель для распознавания изображений, обученная обнаруживать собак и котов, то клиент может отправить ей фотографию и с определенной степенью точности узнать, есть ли на ней собака.

Машинное обучение с точки зрения администраторов кластеров Kubernetes

В этом разделе мы обсудим аспекты, которые необходимо учитывать при развертывании рабочих заданий ML в кластере Kubernetes. Данный материал рассчитан на администраторов кластеров. Самый сложный аспект эксплуатации кластера для команды специалистов по анализу данных — понимание

терминологии. Со временем вам придется выучить большое количество новых терминов, однако не волнуйтесь: вам это по силам. Рассмотрим основные вещи, на которые следует обратить внимание при подготовке кластера для задач машинного обучения.

Обучение модели в Kubernetes

Для обучения моделей ML в Kubernetes нужны как обычные центральные процессоры, так и графические адаптеры (GPU). Обычно чем больше ресурсов выделить, тем быстрее завершится этот процесс. В большинстве случаев для обучения модели достаточно одного сервера, который обладает необходимыми ресурсами. Многие облачные провайдеры предлагают виртуальные машины (ВМ) с несколькими графическими адаптерами, поэтому мы советуем вам до перехода к распределенному обучению масштабировать ваш сервер вертикально, добавляя в него от четырех до восьми GPU. В ходе обучения моделей специалисты по анализу данных используют методику под названием *«оптимизация гиперпараметров»*. Гиперпараметр — это просто значение по умолчанию, которое устанавливается перед началом процесса. Сам подход заключается в выполнении большого количества одинаковых заданий ML с разными гиперпараметрами.

Обучение вашей первой модели в Kubernetes

В данном примере мы обучим модель для классификации изображений с помощью публичного набора данных MNIST, который часто применяется для этих целей.

Для обучения модели нам понадобятся графические адаптеры. Проверим, доступны ли они в вашем кластере Kubernetes. Вывод следующей команды показывает, что в нашем кластере имеется четыре GPU:

```
$ kubectl get nodes -o yaml | grep -i nvidia.com/gpu
      nvidia.com/gpu: "1"
      nvidia.com/gpu: "1"
      nvidia.com/gpu: "1"
      nvidia.com/gpu: "1"
```

Поскольку процесс обучения — пакетное рабочее задание, для его выполнения мы будем использовать стандартный ресурс `Job`. Мы повторим этот процесс 500 раз на одном графическом адаптере. Создайте следующий манифест и сохраните его в локальный файл `mnist-demo.yaml`:

```
apiVersion: batch/v1
kind: Job
```

```
metadata:
  labels:
    app: mnist-demo
  name: mnist-demo
spec:
  template:
    metadata:
      labels:
        app: mnist-demo
    spec:
      containers:
      - name: mnist-demo
        image: lachlanevenson/tf-mnist:gpu
        args: ["--max_steps", "500"]
        imagePullPolicy: IfNotPresent
        resources:
          limits:
           nvidia.com/gpu: 1
      restartPolicy: OnFailure
```

Теперь создайте соответствующий объект в своем кластере Kubernetes:

```
$ kubectl create -f mnist-demo.yaml
job.batch/mnist-demo created
```

Проверьте состояние задания, которое вы создали только что:

```
$ kubectl get jobs
NAME         COMPLETIONS   DURATION   AGE
mnist-demo   0/1           4s         4s
```

Задание, занимающееся обучением, должно значиться в списке pod:

```
$ kubectl get pods
NAME               READY   STATUS     RESTARTS   AGE
mnist-demo-hv9b2   1/1     Running    0          3s
```

Просматривая журнал pod, можно наблюдать за процессом обучения:

```
$ kubectl logs mnist-demo-hv9b2
2019-08-06 07:52:21.349999: I tensorflow/core/platform/cpu_feature_guard.cc:
137] Your CPU supports instructions that this TensorFlow binary was not compiled
to use: SSE4.1 SSE4.2 AVX AVX2 FMA
2019-08-06 07:52:21.475416: I tensorflow/core/common_runtime/gpu/gpu_device.cc:
1030] Found device 0 with properties:
name: Tesla K80 major: 3 minor: 7 memoryClockRate(GHz): 0.8235
pciBusID: d0c5:00:00.0
totalMemory: 11.92GiB freeMemory: 11.85GiB
2019-08-06 07:52:21.475459: I tensorflow/core/common_runtime/gpu/gpu_device.cc:
1120] Creating TensorFlow device (/device:GPU:0) -> (device: 0, name: Tesla
K80, pci bus id: d0c5:00:00.0, compute capability: 3.7)
```

```
2019-08-06 07:52:26.134573: I tensorflow/stream_executor/dso_loader.cc:139]
successfully
opened CUDA library libcupti.so.8.0 locally
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /tmp/tensorflow/input_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/tensorflow/input_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/tensorflow/input_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/tensorflow/input_data/t10k-labels-idx1-ubyte.gz
Accuracy at step 0: 0.1255
Accuracy at step 10: 0.6986
Accuracy at step 20: 0.8205
Accuracy at step 30: 0.8619
Accuracy at step 40: 0.8812
Accuracy at step 50: 0.892
Accuracy at step 60: 0.8913
Accuracy at step 70: 0.8988
Accuracy at step 80: 0.9002
Accuracy at step 90: 0.9097
Adding run metadata for 99
...
```

О завершении обучения можно узнать по состоянию задания:

```
$ kubectl get jobs
NAME         COMPLETIONS   DURATION   AGE
mnist-demo   1/1           27s        112s
```

Для удаления обучающего задания достаточно выполнить следующую команду:

```
$ kubectl delete -f mnist-demo.yaml
job.batch "mnist-demo" deleted
```

Поздравляем! Вы только что выполнили свое первое задание по обучению модели в Kubernetes.

Распределенное обучение в Kubernetes

Распределенное обучение все еще находится на ранней стадии развития, и его сложно оптимизировать. Если ваше задание требует восемь графических адаптеров, то его практически наверняка будет быстрее выполнить на одном узле с восьмью GPU, чем на двух узлах с четырьмя GPU на каждом. К распределенному обучению следует прибегать, только когда для вашей модели не хватает самого крупного из доступных вам серверов. Если вы

уверены в том, что вам необходимо несколько серверов, то очень важно, чтобы вы понимали архитектуру этого процесса. На рис. 14.2 показана распределенная архитектура TensorFlow, которая наглядно демонстрирует, как распределяются модель и ее параметры.

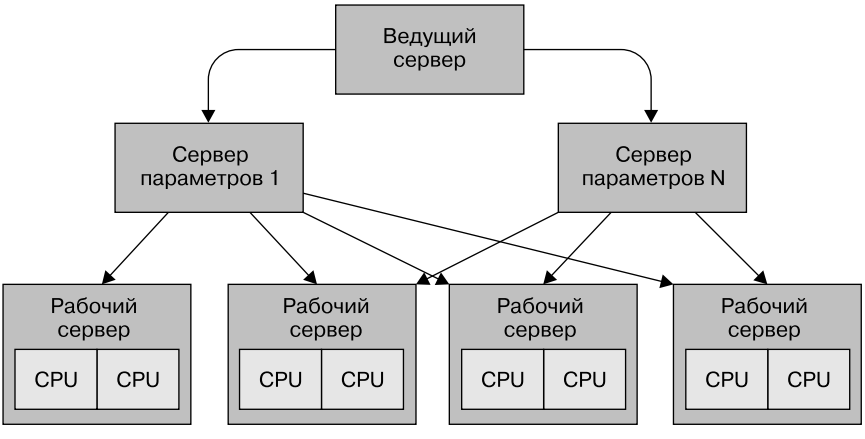


Рис. 14.2. Распределенная архитектура TensorFlow

Требования к ресурсам

Рабочие задания ML требуют специфической конфигурации всевозможных аспектов вашего кластера. Этапы обучения, несомненно, имеют самые высокие требования к ресурсам. Как уже упоминалось ранее, алгоритм ML почти всегда представляет собой пакетное рабочее задание. В частности, у него есть время начала и время завершения. Последнее зависит от того, насколько быстро вам удастся удовлетворить требования модели. Это значит, что процесс обучения почти наверняка можно ускорить за счет масштабирования, однако у данного подхода есть свои узкие места.

Специализированное оборудование

Эффективность обучения и применения модели почти всегда можно повысить с помощью специализированного оборудования. Типичный пример тому — облачные графические адаптеры. Для доступа к ним Kubernetes использует специальные дополнения, предоставляющие информацию о GPU планировщику. Для этого предусмотрен фреймворк «аппаратных»

дополнений, благодаря чему поддержка специфических устройств не требует модификации исходного кода Kubernetes. Эти дополнения обычно имеют вид объектов DaemonSet, которые запускаются на каждом узле и отвечают за предоставление API данных о соответствующих ресурсах. Рассмотрим аппаратное дополнение для Kubernetes от NVIDIA (https://oreil.ly/RgKuz), которое делает доступными графические адаптеры этого производителя. После его установки вы сможете создать следующий pod, и Kubernetes позаботится о том, чтобы он был развернут на узле, имеющем данный ресурс:

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  containers:
    - name: digits-container
      image: nvidia/digits:6.0
      resources:
        limits:
          nvidia.com/gpu: 2 # запрашиваем 2 GPU
```

Аппаратные дополнения не ограничиваются одними лишь графическими адаптерами; их можно использовать для любого специализированного оборудования — например, для программируемых пользователем вентильных матриц (ППВМ) или InfiniBand.

Особенности планирования

Необходимо отметить: платформа Kubernetes не может принимать решения относительно ресурсов, о которых ей ничего не известно. Вы можете заметить, что во время обучения ваши графические адаптеры не работают на полную мощность. Вернемся к предыдущему примеру; в нем указано только количество графических ядер, но мы не знаем, сколько потоков может выполняться на одном ядре. Кроме того, у нас нет информации о том, на какой шине находится графическое ядро, поэтому мы не можем сделать так, чтобы задания, взаимодействующие друг с другом, размещались на одних и тех же узлах. Все эти проблемы могут быть решены в будущих версиях аппаратных дополнений, но пока вам остается лишь догадываться, почему вы не можете добиться 100 % загруженности от мощного графического адаптера, который купили только что. Стоит упомянуть и о том, что вы не можете запросить часть GPU (например, 0,1); это значит, вам не удастся использовать многопоточность, даже если ваш графический адаптер поддерживает ее.

Библиотеки, драйверы и модули ядра

Доступ к специализированному оборудованию обычно требует специальных библиотек, драйверов и модулей ядра. Вам необходимо позаботиться о том, чтобы они были доступны для инструментов внутри контейнеров. Вы можете спросить: «Почему бы их просто не добавить непосредственно в образ контейнера?» Ответ прост: инструменты должны быть совместимы с версией внутренней системы и должным образом сконфигурированы для работы с ней. Отдельные среды выполнения контейнеров, такие как NVIDIA Docker (oreil.ly/Re0Ef), не требуют от вас подключения системных томов к каждому контейнеру. Но то же самое можно реализовать с помощью вебхука доступа. Кроме того, важно понимать, что для доступа к некоторому специализированному оборудованию могут понадобиться контейнеры с повышенными привилегиями, что повлияет на уровень безопасности кластера. Установкой сопутствующих библиотек, драйверов и модулей ядра тоже могут заниматься аппаратные дополнения Kubernetes. Многие из этих дополнений сначала проверяют наличие всех необходимых компонентов и только потом позволяют планировщику Kubernetes использовать ресурсы GPU.

Хранение

Хранение — одна из важных составляющих процесса машинного обучения, определяющих следующие его этапы:

- хранение и распределение набора данных между рабочими узлами во время обучения;
- создание срезов и сохранение моделей.

Хранение и распределение набора данных между рабочими узлами во время обучения

Во время обучения набор данных должен быть доступен для каждого рабочего узла. Доступ к хранилищу следует давать только на чтение, и обычно чем быстрее накопитель, тем лучше. Тип накопителя, который нужно выбирать в качестве хранилищ, почти полностью зависит от размеров набора данных. Блочное хранилище идеально подойдет для данных объемом в сотни мегабайт или гигабайт, но если речь идет о сотнях терабайт, то объектное хранилище может оказаться лучшим решением. От размера и размещения дисков также может зависеть загруженность вашей сети.

Создание срезов и сохранение моделей

Срезы создаются по ходу обучения, а сохранение моделей позволяет применять их в дальнейшем. В обоих случаях к каждому рабочему узлу необходимо подключить хранилище, в которое будут записываться данные. Они обычно хранятся в общем каталоге, а каждый рабочий узел записывает свои результаты в отдельный срез и файл сохранения. Большинство инструментов рассчитаны на то, что срез и сохраненные данные находятся в одном месте и требуют доступ типа `ReadWriteMany`, означающий, что том может быть подключен для чтения и записи ко многим узлам. При использовании томов `PersistentVolume` в Kubernetes необходимо выбрать наиболее подходящую платформу хранения данных. На сайте Kubernetes есть список дополнений, которые предоставляют тома с поддержкой `ReadWriteMany` (oreil.ly/aMjGd).

Организация сети

Этап обучения в рабочем процессе ML оказывает огромное влияние на загруженность сети (особенно при использовании распределенного обучения). Если взять распределенную архитектуру TensorFlow, то большое количество трафика генерируется на двух отдельных стадиях: передача переменных от каждого сервера параметров каждому рабочему узлу и применение градиентов в обратном направлении (см. рис. 14.2). От времени, которое затрачивается на этот обмен данными, напрямую зависит, насколько быстро будет проходить обучение модели. Здесь все просто: чем быстрее, тем лучше (в пределах разумного, конечно). На сегодня большинство серверов и публичных облаков поддерживают сетевые интерфейсы с пропускной способностью 1, 10 и иногда 40 Гбит/с, так что об этом стоит беспокоиться, только имея дело с медленной сетью. Если вам нужна высокая пропускная способность, то можете подумать об использовании InfiniBand.

И хотя в большинстве случаев скорость сетевого соединения все еще выступает ограничивающим фактором, сама доставка данных из ядра в сеть тоже может быть проблемой. Некоторые проекты с открытым исходным кодом пытаются ускорить сетевой трафик за счет использования технологии удаленного прямого доступа к памяти (Remote Direct Memory Access, RDMA) и при этом не требуют модификации рабочих узлов или кода приложения. RDMA позволяют компьютерам, находящимся в одной сети, разделять свою главную память, не прибегая к использованию своих процессоров, кэша и операционных систем. Можете обратить внимание на открытый проект

Freeflow (https://oreil.ly/3RBNS), который, согласно обещаниям разработчиков, обеспечивает высокую производительность обмена данными между контейнерами, объединенными в оверлейную сеть.

Узкоспециализированные протоколы

Организовать машинное обучение в Kubernetes можно с помощью и других специализированных протоколов. Обычно они предназначены только для оборудования определенного поставщика, но цель у них одна: решение проблем с масштабированием распределенного обучения путем устранения тех участков архитектуры, которые быстро становятся узкими местами (таких как серверы параметров). Эти протоколы часто позволяют обмениваться информацией между графическими адаптерами на разных узлах в обход центральных процессоров и операционных систем, помогая более эффективно масштабировать распределенное обучение. Два из них перечислены ниже:

- MPI (Message Passing Interface) — стандартизированный переносимый API для передачи данных между распределенными процессами;
- NCCL (NVIDIA Collective Communications Library) — библиотека компонентов для взаимодействия между GPU с поддержкой разных сетевых топологий.

Машинное обучение с точки зрения специалистов по анализу данных

В предыдущем разделе мы обсудили вещи, на которые необходимо обращать внимание при выполнении машинного обучения в кластере Kubernetes. Но что насчет специалистов по анализу данных? Здесь мы рассмотрим некоторые популярные инструменты, позволяющие использовать Kubernetes для ML даже тем, кто не имеет большого опыта работы с этой платформой.

- *Kubeflow* (www.kubeflow.org) — набор инструментов для организации машинного обучения с использованием стандартных средств Kubernetes. Он содержит несколько инструментов, необходимых для налаживания процесса ML. Jupyter Notebook, конвейеры и стандартные для Kubernetes контроллеры упрощают и облегчают применение Kubernetes в качестве платформы для машинного обучениия.
- *Polyaxon* (polyaxon.com) — инструмент для управления процессами машинного обучения, поддерживающий множество популярных библиотек

и совместимый с любым кластером Kubernetes. Имеет две версии: коммерческую и открытую.

- *Pachyderm* (www.pachyderm.io) — платформа для анализа данных уровня предприятия с богатым набором инструментов для подготовки наборов информации, организации жизненного цикла, ведения версий и построения конвейеров машинного обучения. Имеет коммерческую версию, которую можно развернуть в любом кластере Kubernetes.

Рекомендации по машинному обучению в Kubernetes

Чтобы добиться оптимальной производительности ваших рабочих заданий с машинным обучением, руководствуйтесь следующими рекомендациями.

- Разумное планирование и автомасштабирование. Учитывая, что большинство этапов машинного обучения являются пакетными по своей природе, мы советуем задействовать Cluster Autoscaler. Серверы с поддержкой GPU стоят дорого, поэтому нельзя допускать, чтобы они простаивали без дела. Мы рекомендуем запускать пакетные задания только в нужные вам моменты времени с использованием ограничений и допусков или за счет запланированного масштабирования с помощью Cluster Autoscaler. Таким образом кластер может подстраиваться под нужды рабочих заданий с ML, именно когда это требуется и ни секундой раньше. Что касается ограничений и допусков, официальная документация рекомендует ограничивать узлы, используя в качестве ключа расширенный ресурс. Например, узел с NVIDIA GPU должен быть ограничен так: `Key: nvidia.com/gpu, Effect: NoSchedule`. Кроме того, этот метод позволяет задействовать контроллер доступа `ExtendedResourceToleration`, который автоматически добавляет соответствующие допуски для pod, запрашивающих расширенные ресурсы. Следовательно, пользователям не нужно будет делать это вручную.
- Процесс обучения требует тонкого баланса. Если один участок отработает слишком быстро, то в другом могут возникнуть задержки. Вы постоянно должны следить за своей моделью и корректировать ее при необходимости. Исходя из нашего опыта, можем дать следующий совет: попытайтесь сделать так, чтобы вашим узким местом стал графический адаптер, поскольку это самый дорогой ресурс. Ваши GPU должны испытывать высокую нагрузку. Всегда следите за появлением узких мест; настройте систему мониторинга для отслеживания загруженности графических адаптеров, процессоров, сети и хранилищ данных.

- Кластеры со смешанными рабочими заданиями. Кластеры, выполняющие рутинные бизнес-процессы, можно применять и для машинного обучения. Учитывая, что ML имеет высокие требования к производительности, мы рекомендуем использовать отдельный пул узлов, на которых можно развертывать только рабочие задания с машинным обучением. Это поможет изолировать остальную часть кластера от нагрузок, вызванных рабочими заданиями в данном пуле. Более того, вы можете даже создать несколько пулов с поддержкой GPU, рассчитанные на рабочие задания с разными требованиями к производительности. Мы также рекомендуем включить для этих пулов автоматическое масштабирование узлов. Прежде чем использовать кластеры со смешанными характеристиками, необходимо тщательно разобраться с тем, какое влияние имеют ваши рабочие задания с ML на остальную систему.
- Достижение линейного масштабирования при распределенном обучении. Это святой Грааль в области распределенного обучения моделей. К сожалению, большинство библиотек не масштабируются линейно. В настоящее время для решения этой проблемы прилагаются большие усилия, но пока необходимо понимать, что здесь нельзя обойтись одним лишь увеличением количества аппаратных ресурсов. Как показывает наш опыт, узким местом почти всегда оказывается сама модель, а не инфраструктура, в рамках которой она работает. Но прежде, чем винить модель, следует проверить загруженность GPU, процессора, сети и хранилища данных. Такие открытые инструменты, как Horovod (github.com/horovod/horovod), стремятся улучшить фреймворки распределенного обучения и обеспечить более эффективное масштабирование модели.

Резюме

В этой главе мы собрали очень много материала. Надеемся, теперь вы понимаете, почему Kubernetes является отличной платформой для машинного (и особенно глубокого) обучения, и знаете, на что следует обратить внимание перед развертыванием вашего первого рабочего задания с ML. Рекомендации, перечисленные в данной главе, станут хорошим подспорьем для построения и обслуживания кластеров Kubernetes, рассчитанных на машинное обучение.

ГЛАВА 15

Построение высокоуровневых абстракций на базе Kubernetes

Kubernetes — сложная система. Она упрощает развертывание и администрирование распределенных приложений, но мало чем помогает с их разработкой. Действительно, все новые концепции и артефакты, с которыми приходится взаимодействовать разработчикам, повышают уровень сложности в угоду упрощенного администрирования. В связи с этим во многих средах поверх Kubernetes имеет смысл создавать более высокоуровневые и дружественные к разработчикам абстракции. Кроме того, многим компаниям было бы полезно стандартизировать процессы конфигурации и развертывания приложений, чтобы обеспечить соблюдение общепринятых рекомендаций. Этого можно добиться и за счет предоставления высокоуровневых абстракций, которые будут вынуждать разработчиков следовать данным принципам. Но при этом может потеряться доступ к важным аспектам системы и появиться барьер, который станет ограничивать или усложнять разработку определенных приложений или интеграцию существующих решений. С самого начала развития облачных технологий наблюдается постоянный конфликт между гибкостью инфраструктуры и богатством возможностей, предоставляемых платформой. Проектирование подходящих высокоуровневых абстракций позволяет поддерживать тонкий баланс между этими двумя качествами.

Разные подходы к разработке высокоуровневых абстракций

Существует два основных подхода к разработке высокоуровневых абстракций поверх Kubernetes. Первый — инкапсулировать всю платформу так, чтобы разработчики могли и не знать о том, что находится внутри. Они будут иметь дело с системой, которую предоставляете вы, а Kubernetes превратится в один из аспектов реализации.

Второй вариант заключается в использовании стандартных возможностей расширения Kubernetes. API-сервер Kubernetes довольно гибок и поддерживает динамическое добавление высокоуровневых ресурсов, которые будут существовать наряду со встроенными в платформу объектами. При этом для взаимодействия и с теми и с другими пользователи смогут применять стандартные инструменты. В такой модели расширения основная часть разработки по-прежнему ориентирована на Kubernetes, а дополнительные компоненты понижают уровень сложности и упрощают применение вашей системы.

Как выбрать из представленных двух вариантов наиболее подходящий? В действительности это зависит от того, какие цели вы преследуете, создавая свой слой абстракции. Если вы пытаетесь построить полностью изолированную, интегрированную среду, которая не даст вашим пользователям выйти за рамки дозволенного и с которой будет легко работать, то лучше выбрать первый подход. Хороший пример такого сценария — создание процесса машинного обучения. Здесь все относительно просто. Специалисты по анализу данных, скорее всего, незнакомы с Kubernetes. Основная цель — позволить им быстро выполнять их работу и заниматься своей проблемной областью, не отвлекаясь на распределенные системы. Поэтому создание полноценной обертки вокруг Kubernetes выглядит наиболее разумным решением.

С другой стороны, если ваши высокоуровневые абстракции предназначены для разработчиков (например, чтобы им было проще развертывать Java-приложения), то Kubernetes лучше всего не инкапсулировать, а расширять. Тому есть две причины. Во-первых, предметная область разработчика приложений чрезвычайно широкая. Вам будет очень сложно предвидеть все его нужды и сценарии применения, особенно если технические и бизнес-требования меняются со временем. Вторая причина состоит в том, что разработчику и дальше нужно пользоваться преимуществами экосистемы Kubernetes. Для мониторинга, непрерывной доставки и других задач существует огромное количество облачно-ориентированных инструментов. Расширяя, а не заменяя API Kubernetes, вы сможете продолжать задействовать эти и новые инструменты.

Расширение Kubernetes

Вокруг Kubernetes можно создавать совершенно разные обертки, и описать их все мы просто не в состоянии. А вот инструменты и методики для расширения Kubernetes универсальны, в каком бы проекте вы их ни применяли, поэтому мы уделим им немного внимания.

Расширение кластеров Kubernetes

Расширение кластеров Kubernetes — обширная тема, более полное рассмотрение которой можно найти в таких книгах, как *Managing Kubernetes* (https://oreil.ly/6kUUX) и *Kubernetes: Up and Running* (https://oreil.ly/fdRA3) (O'Reilly). Чтобы не дублировать один и тот же материал, в данном подразделе мы сосредоточимся на использовании стандартных средств расширения Kubernetes. Здесь необходимо понимать, как ресурсы кластера взаимодействуют друг с другом. В этом контексте существует три связанных между собой технических решения. Первое — паттерн проектирования *«Прицеп»* (Sidecar). Он представляет собой вспомогательный контейнер (рис. 15.1) и активно применяется в области межсервисного взаимодействия. Такие контейнеры размещаются рядом с самим приложением и предоставляют различные функции, которые вынесены из основного кода и часто разрабатываются отдельной командой. Например, в mesh-сетях вспомогательный контейнер может отвечать за прозрачную взаимную аутентификацию по TLS (mutual Transport Layer Security, mTLS) в приложении.

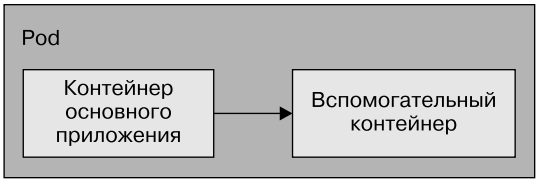


Рис. 15.1. Паттерн проектирования «Прицеп»

Этот паттерн позволяет добавлять новые возможности в пользовательские приложения.

Конечно, вся суть данного подхода состоит в том, чтобы упростить жизнь разработчикам, но если им нужно будет знать об этих вспомогательных контейнерах и уметь их использовать, то это лишь усугубит проблему. К счастью, процесс расширения Kubernetes можно упростить с помощью определенных инструментов. В частности, Kubernetes поддерживает *контроллеры доступа*, перехватывающие запросы к API платформы еще до того, как они попадают во внутреннее хранилище кластера. Вы можете использовать их для проверки и модификации объектов API. В контексте паттерна *«Прицеп»* они позволяют автоматически добавлять вспомогательные контейнеры во все pod, которые создаются в кластере, и, чтобы пользоваться их преимуществами,

разработчикам не обязательно знать об их существовании. Взаимодействие контроллеров доступа с API Kubernetes показано на рис. 15.2.

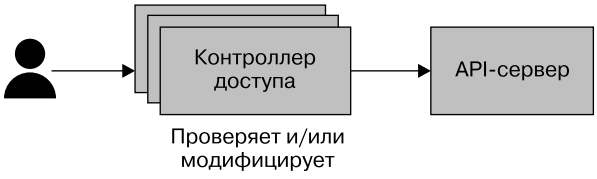


Рис. 15.2. Контроллеры доступа

Помимо добавления вспомогательных контейнеров, контроллеры доступа имеют и другое применение. Они позволяют проверять объекты, передаваемые разработчиками платформе Kubernetes. Например, вы можете реализовать *анализатор*, который будет следить за тем, чтобы pod и другие ресурсы, создаваемые разработчиками, соответствовали общепринятым рекомендациям. Отдельные приложения не резервируют нужные им ресурсы — это распространенная ошибка. В такой ситуации анализатор на основе контроллера доступа может перехватить и отклонить запрос на создание pod. Конечно, при необходимости продвинутые пользователи должны иметь возможность обходить правила анализатора (например, с помощью специальных аннотаций). О важности этого замечания мы поговорим чуть позже.

Итак, мы обсудили то, каким образом можно расширять существующие приложения и принуждать разработчиков к соблюдению общепринятых рекомендаций. Но что насчет добавления высокоуровневых абстракций? Здесь нам пригодятся пользовательские определения ресурсов (custom resource definitions, CRD). CRD позволяют динамически добавлять в кластер Kubernetes новые типы объектов. Например, вы можете определить новый ресурс `ReplicatedService`, который при создании обращается к Kubernetes, чтобы создать объект `Deployment` и сервис. Таким образом, `ReplicatedService` может служить удобной абстракцией для выполнения рутинных действий. Определения CRD обычно создаются в рамках управляющего цикла, который развертывается в кластере в целях управления новыми типами ресурсов.

Расширение пользовательских аспектов Kubernetes

Добавление новых ресурсов — отличный способ расширить возможности кластера, но, чтобы извлечь из этого максимальную пользу, стоит подумать и о расширении пользовательских аспектов Kubernetes. Инструментарий

Kubernetes сам по себе ничего не знает о пользовательских ресурсах и других расширениях, так что работа с ними не отличается особыми гибкостью и удобством. Это можно исправить за счет расширения стандартных утилит командной строки.

В целом для работы с Kubernetes обычно используют утилиту `kubectl`, которую, к счастью, тоже можно расширить. `kubectl` поддерживает двоичные дополнения с именами наподобие `kubectl-foo` (где `foo` — название дополнения). При выполнении команды `kubectl foo ...` запускается соответствующий двоичный файл. Дополнения к `kubectl` позволяют создавать новые пользовательские команды, которые понимают все нюансы ресурсов, добавленных вами в кластер. Вы вольны реализовывать любые возможности, которые вам нужны, задействуя при этом знакомый инструментарий `kubectl`. Особенно ценно здесь то, что разработчикам не придется изучать новый набор инструментов. К тому же вы можете вводить в Kubernetes новые концепции по мере того, как разработчики углубляют свое понимание данной платформы.

Архитектурные аспекты построения новых платформ

В целях повышения продуктивности разработчиков было создано огромное количество платформ. Наблюдая за их успехами и неудачами, вы можете сформировать общий набор методик и рекомендаций, основанный на чужом опыте. Соблюдение этих принципов поможет вам сделать вашу платформу успешной и избежать тупиковых путей развития, у которых нет будущего.

Поддержка экспорта в образ контейнера

Многие платформы дают пользователям возможность загружать вместо полноценного образа контейнера сам код (например, функция в FaaS — Function as a Service (функция как услуга)) или стандартный пакет (такой как JAR-файл в Java). Этот подход отличается простотой и позволяет разработчикам оставаться в знакомой им экосистеме. Платформа сама заботится о контейнеризации приложения.

Но, когда разработчик сталкивается с ограничениями среды программирования, которую вы ему предоставили, возникает проблема. Возможно, ему нужна определенная версия среды выполнения языка, чтобы обойти какую-то

ошибку; или же автоматическая контейнеризация приложения не позволяет ему упаковать дополнительные ресурсы или исполняемые файлы.

Какой бы ни была причина, это серьезный барьер, и его преодоление может оказаться довольно болезненным, поскольку разработчику внезапно необходимо научиться упаковывать свое приложение, хотя в действительности он хотел лишь немного его расширить, чтобы исправить какую-то ошибку или добавить новую возможность.

Но это вовсе не единственный путь. Если вы поддерживаете экспорт среды программирования своей платформы в универсальный контейнер, то разработчику, который взаимодействует с данной платформой, не нужно начинать с чистого листа и дотошно изучать все нюансы контейнеризации. Вместо этого он получает полноценный рабочий образ контейнера, который представляет его текущее приложение (например, функцию и среду выполнения узла). Затем он может адаптировать образ контейнера под свои нужды, внося небольшие изменения. Такой плавный переход от высокоуровневой платформы к низкоуровневой инфраструктуре и постепенное изучение необходимых технологий делает платформу более практичной, поскольку при ее использовании не возникает никаких существенных барьеров.

Поддержка существующих механизмов для обнаружения сервисов и работы с ними

Еще одним распространенным свойством платформ является тот факт, что они развиваются и взаимодействуют с другими системами. Ваша платформа позволит улучшить продуктивность многих разработчиков, но любое реальное приложение будет выходить за пределы ее и низкоуровневых компонентов Kubernetes, охватывая *другие* платформы. Соединения с существующими базами данных или открытыми сервисами, разработанными для Kubernetes, всегда будут частью любого достаточно крупного приложения.

Поскольку такая взаимосвязанность необходима, любая платформа, которую вы разрабатываете, обязательно должна применять и предоставлять стандартные компоненты Kubernetes для работы с сервисами и их обнаружения. Не изобретайте велосипед ради удобства пользователей, иначе ваша система будет неспособна взаимодействовать с внешним миром.

Если вы оформите свой код в виде сервисов Kubernetes, то его сможет использовать любое приложение в вашем кластере, даже если оно работает

в рамках вашей высокоуровневой платформы. Точно так же организация обнаружения сервисов на основе DNS-серверов Kubernetes позволит вашей высокоуровневой платформе взаимодействовать с другими приложениями в кластере, даже если они работают на более низком уровне. У вас может возникнуть соблазн построить нечто более хорошее или простое в использовании, но взаимосвязь между разными платформами — общая черта любого достаточно зрелого и сложного приложения. Вы обязательно пожалеете о решении создать закрытую экосистему.

Рекомендации по созданию прикладных платформ

Платформа Kubernetes предоставляет эффективные инструменты для управления программным обеспечением, но мало чем помогает в разработке приложений. Поэтому в целях предоставления разработчикам возможности действовать более продуктивно и/или упрощения использования Kubernetes часто возникает необходимость в построении более высокоуровневой платформы поверх уже существующей. В этом вам помогут следующие рекомендации.

- Используйте контроллеры доступа для ограничения и изменения API-вызовов к кластеру. Эти контроллеры могут проверять ресурсы Kubernetes и запрещать их развертывание в случае несоблюдения определенных требований. Контроллеры также способны модифицировать эти ресурсы, добавляя к ним вспомогательные контейнеры или внося какие-либо другие изменения, о которых пользователь может и не догадываться.
- Задействуйте дополнения к `kubectl` для расширения взаимодействия пользователей с Kubernetes за счет добавления новых команд в уже знакомую утилиту командной строки. В редких случаях может возникнуть необходимость в создании специализированного инструмента.
- При создании платформы поверх Kubernetes основательно обдумайте ее применение и то, как будут меняться нужды ее пользователей. Удобство и простота в использовании — несомненно, то, к чему нужно стремиться. Но если для работы с вашей платформой пользователям необходимо переписывать все, что находится за ее пределами, то это неизбежно приведет к разочаровывающим и неудачным результатам.

Резюме

Kubernetes — фантастический инструмент, упрощающий развертывание и администрирование ПО, но, к сожалению, с точки зрения разработчиков, данная среда не всегда дружественна и продуктивна. В связи с этим поверх Kubernetes часто создают высокоуровневые платформы, которые лучше подходят для среднего разработчика. В этой главе мы описали несколько подходов к проектированию таких высокоуровневых систем и дали краткий обзор основных средств для расширения, присутствующих в Kubernetes. В конце вы познакомились с рекомендациями и архитектурными принципами, основанными на наших наблюдениях за другими платформами, построенными поверх Kubernetes. Надеемся, это поможет вам в проектировании собственной платформы.

ГЛАВА 16

Управление состоянием

На ранних этапах развития систем оркестрации контейнеров роль рабочих заданий обычно играли приложения, которые либо не имели своего состояния (stateless), либо хранили его во внешних сервисах. Это было обусловлено тем, что контейнеры считались очень недолговечными и оркестрация внутреннего хранилища, необходимого для поддержания согласованного состояния, была бы как минимум затруднительной. Но со временем потребность в контейнерных рабочих заданиях с сохранением состояния (stateful) стала реальностью, и в отдельных случаях этот подход может быть более производительным. Платформа Kubernetes прошла длинный путь в данном направлении и теперь позволяет не только подключать тома хранилищ к pod, но и управлять этими томами с помощью стандартных средств. Это стало важным элементом оркестрации рабочих заданий, которым нужно хранить свое состояние.

Если бы подключения внешних томов к контейнерам было достаточно, то в Kubernetes существовало бы куда больше масштабных приложений с сохранением состояния. Но в реальности это самый простой случай. Большинство приложений, которым нужно хранить свое состояние даже после отказа узла, представляют собой сложные информационные системы, такие как реляционные базы данных, хранилища типа «ключ — значение» и развитые платформы для работы с документами. Проекты подобного рода требуют более тесной координации в контексте взаимодействия своих компонентов, идентификации членов кластера и определения порядка, в котором эти члены появляются и исчезают в системе.

Эта глава посвящена общепринятым рекомендациям по управлению состоянием, начиная с простых методик, таких как сохранение файлов на общий сетевой диск, и заканчивая сложными системами наподобие MongoDB, MySQL и Kafka. Отдельный небольшой раздел выделен специально для обзора нового решения под названием Operator, которое не только расширяет стандартные средства Kubernetes, но и позволяет

оформлять бизнес-логику и код приложений в виде пользовательских контроллеров, упрощая тем самым администрирование сложных систем для управления данными.

Тома и их подключение

Не всякое рабочее задание, которому нужно как-то хранить свое состояние, является сложной базой данных или очередью с высокой пропускной способностью. Контейнеризованным приложениям зачастую необходимо записывать и читать информацию в определенных каталогах, и они рассчитывают на их существование. Возможность сохранять данные в томе, который может быть прочитан контейнером в pod, была рассмотрена в главе 5; однако данные, подключенные с помощью объектов `ConfigMap` или `Secret`, обычно доступны только для чтения. В этой главе мы сосредоточимся на предоставлении томов, которые поддерживают запись и могут справиться с отказом контейнера или, что еще лучше, pod.

Любая популярная среда выполнения контейнеров, такая как Docker, rkt, CRI-O и даже Singularity, позволяет подключать тома, привязанные к внешним системам хранения данных. В простейшем случае роль внешнего хранилища может играть участок памяти, путь на сервере, на котором выполняется контейнер, или сетевая файловая система, такая как NFS, Glusterfs, CIFS или Ceph. Вам, наверное, интересно, зачем все это нужно? Представьте старое приложение, написанное для ведения журнала на локальном диске. Разместить его в Kubernetes можно множеством разных способов; например, вы можете обновить его код так, чтобы оно выводило журнальные записи в `stdout` или `stderr`, откуда их будет доставать вспомогательный контейнер и записывать во внешний источник, используя разделяемый том pod или журнальный механизм узла, способный считывать из тома журнал контейнера. Чтобы реализовать последний вариант подключения тома к контейнеру, можно выполнить стандартный для Kubernetes метод `hostPath`, как показано ниже:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-webserver
spec:
  replicas: 3
  selector:
```

```
    matchLabels:
      app: nginx-webserver
  template:
    metadata:
      labels:
        app: nginx-webserver
    spec:
      containers:
      - name: nginx-webserver
        image: nginx:alpine
        ports:
        - containerPort: 80
        volumeMounts:
          - name: hostvol
            mountPath: /usr/share/nginx/html
      volumes:
        - name: hostvol
          hostPath:
            path: /home/webcontent
```

Рекомендации по обращению с томами

- Старайтесь подключать тома только к pod, контейнерам которых нужно разделять данные (например, с помощью таких паттернов проектирования, как «Адаптер» или «Посол» (Ambassador)). В таких случаях используйте `emptyDir`.
- Используйте `hostPath`, когда к данным должны обращаться агенты или сервисы уровня узла.
- Попытайтесь определить, какие сервисы сохраняют важные журнальные записи и события на локальный диск, и по возможности перенаправьте эти данные в `stdout` или `stderr`, чтобы система агрегации, интегрированная с Kubernetes, могла собирать их в виде потоков, а не путем привязки томов.

Хранение данных в Kubernetes

Примеры, представленные ранее, подразумевали подключение томов к контейнерам и pod, что является одной из базовых возможностей контейнерной среды. Но если вы хотите решить данную задачу должным образом, вам нужно позволить Kubernetes управлять хранилищем, на котором основан том. Это позволит справляться с более динамическими ситуациями, в которых

pod создаются и удаляются по мере необходимости, но хранилище всегда остается доступным, независимо от того, где они размещаются. Для управления хранилищами pod в Kubernetes предусмотрено два отдельных API: `PersistentVolume` и `PersistentVolumeClaim`.

PersistentVolume

`PersistentVolume` — это своеобразный диск, который может лежать в основе любого тома, подключаемого к pod. Его время жизни определяется отдельной политикой, не зависящей от жизненного цикла pod, использующей том. Kubernetes поддерживает динамически и статически определяемые тома. Для динамического создания тома необходимо наличие в Kubernetes соответствующего класса хранилища, `StorageClass`. Том `PersistentVolume` можно создавать в разного рода кластерах, но pod сможет использовать его только в случае, если у него есть подходящая заявка на выделение, `PersistentVolumeClaim`. Сами тома основаны на отдельных дополнениях, большое количество которых встроено непосредственно в Kubernetes; каждое дополнение имеет свои собственные конфигурационные параметры:

```
apiVersion: v1
kind: PersistentVolume
metadata:
name: pv001
labels:
  tier: "silver"
spec:
capacity:
  storage: 5Gi
accessModes:
- ReadWriteMany
persistentVolumeReclaimPolicy: Recycle
storageClassName: nfs
mountOptions:
  - hard
  - nfsvers=4.1
nfs:
  path: /tmp
  server: 172.17.0.2
```

PersistentVolumeClaim

Заявка на выделение (`PersistentVolumeClaim`) — это механизм, с помощью которого pod может описать свои требования к хранилищу. Если pod ссылается на такую заявку и в кластере есть объект `PersistentVolume`, отвечающий

заданным требованиям, то Kubernetes выделит ему соответствующий том. В заявке необходимо указать как минимум размер и режим доступа, но вы также можете определить нужный вам класс хранилища (`StorageClass`). Для выделения томов, отвечающих определенным критериям, можно использовать и селекторы:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClass: nfs
    accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
  selector:
    matchLabels:
      tier: "silver"
```

Заявка, приведенная выше, соответствует ранее созданному тому `PersistentVolume`, поскольку у них совпадают классы хранилища, селекторы, размеры и режимы доступа.

Kubernetes сопоставит `PersistentVolume` с заявкой и свяжет их вместе. Чтобы использовать том, в спецификации pod достаточно указать имя заявки, как показано ниже:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-webserver
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-webserver
  template:
    metadata:
      labels:
        app: nginx-webserver
  spec:
    containers:
    - name: nginx-webserver
      image: nginx:alpine
      ports:
      - containerPort: 80
      volumeMounts:
```

```
      - name: hostvol
        mountPath: /usr/share/nginx/html
  volumes:
    - name: hostvol
      persistentVolumeClaim:
        claimName: my-pvc
```

Классы хранилищ

Вместо того чтобы заранее и вручную определять тома `PersistentVolume`, администраторы могут создавать объекты `StorageClass` с указанием дополнения, которое нужно использовать, всех необходимых параметров подключения и конфигурации, которая будет распространяться на все тома этого класса. Таким образом, вы сможете указать в своей заявке на выделение определенного класса хранилища, и Kubernetes динамически создаст том, основанный на параметрах и конфигурации `StorageClass`:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
name: nfs
provisioner: cluster.local/nfs-client-provisioner
parameters:
  archiveOnDelete: True
```

Kubernetes также позволяет администраторам создать класс хранилища по умолчанию, используя дополнение доступа `DefaultStorageClass` (которое должно быть активировано в API-сервере). В этом случае данный класс будет применяться для всех томов, в которых явно не задано поле `storageClassName`. Часть облачных провайдеров предоставляют класс хранилища по умолчанию, привязанный к самому дешевому доступному типу хранилищ.

Интерфейс хранилищ для контейнеров и FlexVolume

Дополнения CSI (Container Storage Interface — интерфейс хранилищ для контейнеров) и FlexVolume, не входящие в основной репозиторий Kubernetes, позволяют поставщикам систем хранения данных создавать собственные хранилища, не дожидаясь, пока их код включат в кодовую базу Kubernetes (как в случае с большинством других дополнений для работы с томами).

Дополнения CSI и FlexVolume развертываются в кластерах Kubernetes администраторами и при необходимости могут обновляться поставщиками соответствующих хранилищ для добавления новых возможностей.

Цель проекта CSI сформулирована на его GitHub-странице (oreil.ly/AuMgE):

> «Создать отраслевой стандарт, который позволит поставщикам хранилищ разрабатывать дополнения, способные работать с целым рядом систем оркестрации контейнеров».

Интерфейс FlexVolume — традиционное средство, с помощью которого поставщик хранилища может добавлять новые возможности. Для его работы на каждом узле кластера следует установить специальные драйверы. Это, в сущности, исполняемый файл, который должен присутствовать на каждом сервере. Это основной недостаток FlexVolume, особенно с точки зрения провайдеров управляемых сервисов, так как прямой доступ к серверам считается плохим тоном и практически исключает использование ведущих узлов. Дополнение CSI решает данную проблему; оно обладает теми же возможностями, но для его применения в кластере достаточно развернуть всего лишь один pod.

Рекомендации по использованию хранилищ в Kubernetes

Облачно-ориентированные принципы проектирования приложений максимально поощряют отказ от хранения состояния; но с популяризацией контейнерных сервисов возникла нужда в постоянном хранении данных. Перечисленные ниже рекомендации помогут вам сформировать эффективный подход к выбору подходящих реализаций хранилищ данных в архитектуре ваших приложений.

- По возможности включите дополнение доступа `DefaultStorageClass` и определите класс хранилища по умолчанию. В Helm многие чарты приложений, которым требуются тома `PersistentVolume`, используют по умолчанию класс `default`, что позволяет устанавливать эти приложения без излишних модификаций.
- При проектировании архитектуры кластера (с локальным или облачным размещением) учитывайте взаимное размещение и взаимодействие вычислительного слоя и слоя хранения данных; используйте подходящие метки как для узлов, так и для томов `PersistentVolume` и размещайте данные и рабочие задания максимально близко друг к другу с помощью механизма принадлежности. Вряд ли вам захочется, чтобы pod на узле в зоне А подключал том, прикрепленный к узлу в зоне Б.

- Внимательно относитесь к выбору рабочих заданий, которым нужно хранить свое состояние на диске. Можно ли их заменить внешним сервисом, таким как база данных, или, если вы имеете дело с облаком, управляемым сервисом, совместимым с API, которые вы уже используете, — например, SaaS-решениями для MongoDB или MySQL?
- Определите, каких усилий потребует модификация кода приложения для того, чтобы оно больше не сохраняло свое состояние.
- Платформа Kubernetes отслеживает и подключает тома по мере развертывания рабочих заданий, но все еще не умеет обеспечивать их избыточность и резервное копирование хранящихся в них данных. В спецификации CSI предусмотрен API, с помощью которого поставщики могут применять стандартные технологии для создания резервных копий, если таковые поддерживаются внутренним хранилищем.
- Убедитесь в том, что данные, которые будет хранить том, имеют подходящий жизненный цикл. По умолчанию используются динамически выделяемые тома PersistentVolume, удаляемые из внутреннего хранилища при уничтожении pod. Для чувствительной информации и данных, которые могут применяться в судебной экспертизе, политика утилизации должна предусматривать удаление тома.

Приложения с сохранением состояния

Вопреки общему мнению, поддержка приложений с сохранением состояния, таких как MySQL, Kafka, Cassandra, появилась в Kubernetes на самых ранних этапах развития. Но в те дни она вызывала много сложностей и была нацелена в основном на мелкие приложения, а для обеспечения масштабируемости и устойчивости требовалось много усилий.

Сначала разберемся с тем, как типичный объект `ReplicaSet` развертывает и управляет pod и как это затрудняет работу традиционных приложений, хранящих свое состояние:

- pod в `ReplicaSet` горизонтально масштабируются и получают случайные имена при развертывании;
- при уменьшении количества pod в `ReplicaSet` они удаляются произвольным образом;

- pod в `ReplicaSet` никогда не вызываются напрямую по имени или IP-адресу, а только через связанный с ними сервис;
- pod в `ReplicaSet` можно в любой момент перезапустить и переместить на другой узел;
- pod в `ReplicaSet` связаны с томом `PersistentVolume` только с помощью заявки, но эта заявка в случае перемещения на другой узел может быть перенята новым pod с новым именем.

Те, у кого есть поверхностные знания систем управления данными в кластере, уже могут заметить проблемы с pod на основе `ReplicaSet`, которые обладают данными характеристиками. Представьте, что pod с актуальной и доступной для изменения копией базы данных внезапно удалили! Это, несомненно, вызвало бы настоящий хаос.

Большинство новичков в мире Kubernetes считают, что приложения, входящие в `StatefulSet`, являются базами данных. Но это совсем не так — в том смысле, что платформа Kubernetes не имеет никакого представления о типе развертываемых приложений. Она не знает, что вашей базе данных необходимо проводить процесс выбора ведущего сервера, не может организовать репликацию членов `StatefulSet` и вообще не догадывается, что это база данных. Тем не менее `StatefulSet` может помочь со многими из этих аспектов.

Объекты StatefulSet

`StatefulSet` упрощает работу с приложениями, рассчитанными на более надежное поведение узлов и pod. Если взглянуть на список характеристик типичных pod в `ReplicaSet`, становится ясно, что `StatefulSet` предлагает совершенно противоположное. Этот ресурс появился в Kubernetes 1.3 и изначально назывался `PetSet`; он был призван удовлетворить некоторые критически важные потребности, связанные с планированием приложений, хранящих свое состояние, включая сложные системы для управления данными, и управлением этими приложениями.

- Pod в `StatefulSet` горизонтально масштабируются и получают последовательные имена с порядковыми номерами. Пока новый pod не станет полностью готовым к работе (то есть пока не будут пройдены проверки работоспособности и/или готовности), следующая реплика в `StatefulSet` не добавляется.

- Pod в `StatefulSet` находятся за неуправляемым сервисом, и к ним можно обращаться отдельно по их именам.
- Pod, которым нужно подключить том, должны использовать заранее определенный шаблон `PersistentVolume`. Запрашиваемые тома не удаляются вместе со `StatefulSet`.

Спецификация `StatefulSet` очень похожа на объект `Deployment`, если не считать определение сервиса и шаблон `PersistentVolume`. Неуправляемый сервис должен быть создан в первую очередь; он будет использоваться для обращения к отдельным pod. Неуправляемый сервис отличается от обычного только тем, что он не занимается обычной балансировкой нагрузки:

```
apiVersion: v1
kind: Service
metadata:
  name: mongo
  labels:
    name: mongo
spec:
  ports:
  - port: 27017
    targetPort: 27017
  clusterIP: None # так создается неуправляемый сервис
  selector:
    role: mongo
```

Определение `StatefulSet` имеет лишь несколько отличий от спецификации `Deployment`:

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: mongo
spec:
  serviceName: "mongo"
  replicas: 3
  template:
    metadata:
      labels:
        role: mongo
        environment: test
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: mongo
          image: mongo:3.4
```

```
        command:
          - mongod
          - "--replSet"
          - rs0
          - "--bind_ip"
          - 0.0.0.0
          - "--smallfiles"
          - "--noprealloc"
        ports:
          - containerPort: 27017
        volumeMounts:
          - name: mongo-persistent-storage
            mountPath: /data/db
      - name: mongo-sidecar
        image: cvallance/mongo-k8s-sidecar
        env:
          - name: MONGO_SIDECAR_POD_LABELS
            value: "role=mongo,environment=test"
volumeClaimTemplates:
- metadata:
    name: mongo-persistent-storage
    annotations:
      volume.beta.kubernetes.io/storage-class: "fast"
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 2Gi
```

Проект Operator

Объект `StatefulSet`, несомненно, сыграл важную роль в появлении в Kubernetes рабочих заданий со сложными системами хранения данных. Единственная реальная проблема (как уже отмечалось ранее) — Kubernetes не знает о том, что именно выполняется внутри `StatefulSet`. При использовании этого объекта придется тщательно продумать такие операции, как резервное копирование, отработка отказа, обновление, а также регистрация ведущего узла и новых реплик.

Когда платформа Kubernetes начинала развиваться, инженер по мониторингу из компании CoreOS создал для нее новую разновидность облачно-ориентированного программного обеспечения под названием Operator. Изначальный замысел состоял в том, чтобы выделить данные, относящиеся к конкретной предметной области и необходимые для выполнения конкретного приложения, в отдельный контроллер, который расширяет Kubernetes. Представьте,

что вы разрабатываете контроллер `StatefulSet` для развертывания, масштабирования, обновления, резервного копирования и общего обслуживания Cassandra или Kafka. Одни из первых операторов, созданных в рамках проекта Operator, предназначались для etcd и Prometheus; последний использует базу данных на основе временных рядов для хранения метрик. Операторы берут на себя создание, резервное копирование и восстановление конфигурации для серверов Prometheus и etcd. Это, в сущности, новые ресурсы Kubernetes, аналогичные pod и объектам `Deployment`.

До недавних пор операторы представляли собой одноразовые инструменты, разрабатывавшиеся для конкретных приложений инженерами по мониторингу или поставщиками ПО. Но в середине 2018 года компания RedHat создала проект Operator Framework — набор инструментов, включающий в себя SDK с диспетчером жизненного цикла и модули, реализующие такие возможности, как сбор метрик, каталог операторов и функции реестра. Операторы предназначены не только для приложений, которые хранят свое состояние, но благодаря логике своих пользовательских контроллеров они лучше подходят для сложных систем управления данными.

Проект Operator все еще является восходящей технологией в экосистеме Kubernetes, однако он постепенно упрочняет свое положение среди многих поставщиков систем управления данными, облачных провайдеров и инженеров по мониторингу во всем мире, которые хотят воплотить свой опыт администрирования сложных распределенных приложений в Kubernetes. С актуальным списком проверенных операторов можно ознакомиться на сайте OperatorHub (operatorhub.io).

Рекомендации по использованию StatefulSet и Operator

`StatefulSet` и операторы будут полезны крупным распределенным приложениям, которым нужно хранить свое состояние и, возможно, выполнять сложные операции, связанные с администрированием и конфигурацией. Проект Operator все еще активно развивается и пользуется сильной поддержкой сообщества, поэтому приведенные ниже рекомендации основаны на возможностях, имевшихся у него на момент публикации.

- Решение об использовании `StatefulSet` следует принимать с осторожностью, поскольку приложения, хранящие свое состояние, обычно нуждаются в таком уровне управления, который пока нельзя должным

образом обеспечить с помощью средств оркестрации (то, как планируется восполнить этот недостаток Kubernetes, обсуждается в подразделе «Проект Operator» на с. 267).

- Неуправляемый сервис, который нужен для обращения к pod в `StatefulSet` как к отдельным узлам, не создается автоматически. Вы должны создать его сами на этапе развертывания.
- Потребность в порядковых именах и предсказуемом масштабировании вовсе не означает, что приложению нужно назначить том `PersistentVolume`.
- Если узел в кластере перестает отвечать, то его pod, входящие в `StatefulSet`, не удаляются автоматически; вместо этого они с некоторой отсрочкой входят в состояние `Terminating` или `Unkown`. Избавиться от них можно только путем удаления объекта узла из кластера; таким образом, kubelet снова заработает и вы сможете удалить pod напрямую или принудительно через Operator. К принудительному удалению следует прибегать в последнюю очередь и с большой осторожностью, чтобы узел с этими pod не был заново активирован, иначе вы получите в одном кластере два набора pod с одинаковыми именами. Принудительное удаление можно выполнить с помощью команды `kubectl delete pod nginx-0 --grace-period=0 --force`.
- Pod может оставаться в состоянии `Unknown` даже после принудительного удаления. Чтобы стереть запись о нем и заставить контроллер `StatefulSet` создать вместо него новый экземпляр, отправьте API-серверу команду `patch`: `kubectl patch pod nginx-0 -p '{"meta data":{"finalizers":null}}'`.
- Если вы имеете дело со сложной системой для управления данными, основанными на процессах выбора ведущего узла или подтверждения репликации, то используйте хук `preStop`; он поможет вам закрыть любые соединения, ускорить выбор ведущего узла или подтвердить синхронизацию данных перед удалением pod с помощью операции отложенной остановки.
- Если вы имеете дело со сложной системой управления данными, хранящей свое состояние, то вам, возможно, стоит поискать подходящий оператор, который поможет администрировать компоненты с более сложным жизненным циклом. Если данную систему разрабатывает ваша организация, то подумайте о том, чтобы упаковать ее в виде оператора, — это может сделать ее более управляемой. Примеры ищите в CoreOS Operator SDK (coreos.com/operators).

Резюме

Большинство организаций, которые проводят контейнеризацию своих проектов, оставляют без изменений приложения, хранящие состояние. Чем больше облачно-ориентированных систем развертываются с помощью управляемых решений на основе Kubernetes, тем более проблемным становится управление данными. Приложения, хранящие свое состояние, требуют к себе особого внимания, но благодаря появлению `StatefulSet` и проекта Operator их все чаще можно встретить в кластерах. За счет подключения томов к контейнерам операторы могут скрыть от разработчиков приложений особенности системы хранения данных. Тем не менее приложения, которые хранят свое состояние, такие как базы данных, являются сложными распределенными системами и требуют тщательной оркестрации с помощью стандартных компонентов Kubernetes наподобие pod, объектов `ReplicaSet`, `Deployment` и `StatefulSet`; с другой стороны, операторы знают о том, с какими приложениями они работают, и встроенные в Kubernetes API помогают им выводить эти системы на уровень промышленных кластеров.

ГЛАВА 17

Контроль доступа и авторизация

Управление доступом к API Kubernetes не только позволяет сделать ваш кластер безопасным, но также гарантирует, что ко всем его пользователям, рабочим заданиям и компонентам можно применять одни и те же политики и средства управления. В этой главе мы покажем вам, как с помощью контроллеров доступа и модулей авторизации включать определенные возможности и адаптировать их под свои нужды.

На рис. 17.1 видно, как и где происходят управление доступом и авторизация. Здесь показан весь путь прохождения запроса через API-сервер Kubernetes, вплоть до сохранения объекта в хранилище (если он не был отклонен).

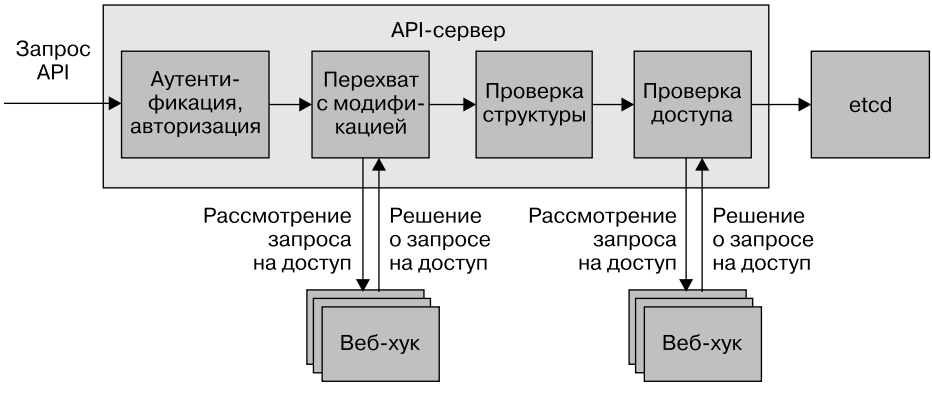


Рис. 17.1. Путь прохождения запроса API

Контроль доступа

Задумывались ли вы когда-нибудь о том, как получается, что в случае определения ресурса в несуществующем пространстве это пространство создается автоматически? Или как выбирается класс хранилища по умолчанию? За все это отвечают малоизвестные компоненты, которые называются

контроллерами доступа. В данном разделе мы поговорим о том, как с помощью этих контроллеров реализовать от имени пользователя общепринятые методики, касающиеся серверной стороны Kubernetes, и как контроль доступа может влиять на сценарии применения вашего кластера.

Что такое контроллеры доступа

Контроллеры доступа находятся на пути следования запросов, направленных к API-серверу Kubernetes, сразу за этапами аутентификации и авторизации. Они используются для проверки и/или модификации объекта запроса до его попадания в хранилище. Контроллер `MutatingAdmissionWebhook` может изменять объекты, которые через него проходят, а `ValidatingAdmissionWebhook` может их только проверять.

Почему они важны

Учитывая то, что контроллеры доступа находятся на пути всех запросов к API-серверу, их можно использовать разнообразными способами. В целом их область применения можно разделить на три категории.

- *Политики и средства управления*. Контроллеры доступа позволяют применять политики, обеспечивая соблюдение бизнес-требований, например:
  - в пространстве имен `dev` можно использовать только внутренние облачные балансировщики нагрузки;
  - у всех контейнеров в pod должны быть установлены лимиты на ресурсы;
  - всем ресурсам должны назначаться заранее заданные стандартные метки или аннотации, чтобы с ними можно было работать с помощью существующих инструментов;
  - все ресурсы Ingress должны использовать только HTTPS; подробнее о том, как в этом контексте применять веб-хуки доступа, можно прочитать в главе 11.
- *Безопасность*. Контроллеры доступа можно задействовать для согласованного обеспечения безопасности на уровне всего кластера. В качестве устоявшегося примера можно привести контроллер доступа `PodSecurityPolicy`; он позволяет контролировать особенно важные с точки зрения безопасности поля спецификации pod и, к примеру, запрещать развертывание

привилегированных контейнеров или использование определенных путей в файловой системе узла. Веб-хуки доступа позволяют вводить более гибкие или нестандартные правила безопасности.

- *Управление ресурсами*. Контроллеры доступа могут проверять соблюдение пользователями вашего кластера общепринятых рекомендаций, например:
  - сделать так, чтобы все внутренние полностью определенные доменные имена (fully qualified domain names, FQDN) имели конкретный суффикс;
  - исключить дублирование FQDN;
  - у всех контейнеров в pod должен быть лимит на ресурсы.

Типы контроллеров доступа

Существует два вида контроллеров доступа: *стандартные* и *динамические*. Первые встроены в API-сервер и поставляются в виде дополнений вместе с каждым выпуском Kubernetes; их нужно конфигурировать во время запуска API-сервера. Динамические контроллеры конфигурируются на этапе выполнения и разрабатываются вне основной кодовой базы Kubernetes; к данной категории относятся только веб-хуки доступа, которые принимают соответствующие запросы через обратные HTTP-вызовы.

В стандартную поставку Kubernetes входит более 30 контроллеров доступа. Чтобы их включить, API-серверу нужно передать следующий флаг:

```
--enable-admission-plugins
```

Многие стандартные функции Kubernetes используют определенные встроенные контроллеры доступа, поэтому по умолчанию рекомендуется указывать следующие параметры:

```
--enable-admissionplugins=NamespaceLifecycle,LimitRanger,ServiceAccount,
DefaultStorage-Class,DefaultTolerationSeconds,MutatingAdmissionWebhook,
ValidatingAdmissionWebhook,Priority,ResourceQuota,PodSecurityPolicy
```

Список всех стандартных контроллеров доступа и их назначение можно найти в документации Kubernetes.

Вы уже могли заметить в вышеприведенном списке следующие элементы: `MutatingAdmissionWebhook`, `ValidatingAdmissionWebhook`. Эти встроенные контроллеры доступа сами по себе не реализуют никакой логики; они используются для конфигурации конечных точек веб-хуков, через которые проходят объекты с запросами доступа.

Конфигурация веб-хуков доступа

Как уже упоминалось ранее, одно из преимуществ веб-хуков доступа заключается в том, что они конфигурируются динамически. Важно понимать, как эффективно конфигурировать эти веб-хуки, поскольку режимы согласованности и отказа имеют определенные последствия и нюансы.

В следующем фрагменте кода показан манифест ресурса `ValidatingWebhookConfiguration`, в котором определяется проверяющий веб-хук доступа. В комментариях подробно описывается назначение каждого поля:

```
apiVersion: admissionregistration.k8s.io/v1beta1
  kind: ValidatingWebhookConfiguration
  metadata:
    name: ## имя ресурса
  webhooks:
  - name: ## имя веб-хука доступа, которое будет видеть пользователь
          ## при отклонении какого-либо запроса
    clientConfig:
      service:
        namespace: ## пространство имен, в котором находится веб-хук доступа
        name: ## имя сервиса, используемого для подключения к веб-хуку доступа
      path: ## URL веб-хука
    caBundle: ## файл CA-Bundle в кодировке PEM, который будет
              ## использоваться для проверки серверного сертификата веб-хука
  rules: ## описывает операции, в ходе которых API-сервер должен
         ## передавать ресурсы/подресурсы этому веб-хуку
  - operations:
    - ## определенная операция, заставляющая API-сервер вызвать этот
      ## веб-хук (например, create, update, delete, connect)
    apiGroups:
    - ""
    apiVersions:
    - "*"
    resources:
    - ## имена отдельных ресурсов (Deployment, Service, Ingress  и т. д.)
  failurePolicy: ## определяет то, как будут обрабатываться проблемы
                 ## с доступом или нераспознанные ошибки (может быть
                 ## равно либо Ignore, либо Fail)
```

Для полноты картины рассмотрим также манифест ресурса `MutatingWebhookConfiguration`, который определяет изменяющий веб-хук доступа. Здесь тоже приводится подробное описание всех полей:

```
apiVersion: admissionregistration.k8s.io/v1beta1
  kind: MutatingWebhookConfiguration
  metadata:
```

```
  name: ## имя ресурса
webhooks:
- name: ## имя веб-хука доступа, которое будет видеть пользователь
        ## при отклонении какого-либо запроса
  clientConfig:
    service:
      namespace: ## пространство имен, в котором находится веб-хук доступа
      name: ## имя сервиса, используемого для подключения к веб-хуку доступа
    path: ## URL веб-хука
  caBundle: ## файл CA-Bundle в кодировке PEM, который будет
            ## использоваться для проверки серверного сертификата веб-хука
rules: ## описывает операции, в ходе которых API-сервер должен
       ## передавать ресурсы/подресурсы этому веб-хуку
- operations:
  - ## определенная операция, заставляющая API-сервер вызвать
     ## этот веб-хук (например, create, update, delete, connect)
  apiGroups:
  - ""
  apiVersions:
  - "*"
  resources:
  - ## имена отдельных ресурсов (Deployment, Service, Ingress и т. д.)
failurePolicy: ## определяет то, как будут обрабатываться проблемы
               ## с доступом или нераспознанные ошибки (может быть
               ## равно либо Ignore, либо Fail)
```

Как вы могли заметить, оба эти ресурса практически идентичны, если не считать поля `kind`. Однако на серверной стороне есть одно отличие: `MutatingWebhookConfiguration` позволяет веб-хуку доступа возвращать модифицированные объекты запросов, а `ValidatingWebhookConfiguration` — нет. Тем не менее вы вполне можете использовать `MutatingWebhookConfiguration` только для проверки, хотя при этом возникают некоторые вопросы, касающиеся безопасности. Мы советуем вам руководствоваться *принципом минимальных привилегий*.

Вы, наверное, уже задумывались о том, «что произойдет, если в поле resources объекта ValidatingWebhookConfiguration или MutatingWebhookConfiguration указать ValidatingWebhookConfiguration или MutatingWebhookConfiguration?» Можете быть спокойны: эти веб-хуки никогда не вызываются при обработке запросов доступа для объектов ValidatingWebhookConfiguration и MutatingWebhookConfiguration. И тому есть хорошая причина, ведь вряд ли вам захочется случайно поместить кластер в состояние, из которого его нельзя восстановить.

Рекомендации по использованию контроллеров доступа

Итак, мы рассмотрели богатые возможности контроллеров доступа. Вот несколько практических рекомендаций, которые помогут вам использовать их максимально эффективно.

- Порядок следования дополнений доступа в параметре `--enable-admission-plugins`, который передается API-серверу, не имеет никакого значения. В ранних версиях Kubernetes он определял, в каком порядке выполняется обработка запросов. В версиях, поддерживаемых в настоящий момент, это больше не так. С другой стороны, порядок следования веб-хуков доступа играет определенную (хоть и незначительную) роль, вследствие чего важно, чтобы вы понимали путь, который в этом случае проходит запрос. Принятие и отклонение запросов выполняется с помощью логического И. То есть если любой из веб-хуков доступа отклонит запрос, то весь процесс проверки будет признан неудачным и пользователь получит ошибку. Необходимо также отметить, что изменяющие контроллеры доступа всегда выполняются перед проверяющими. В этом есть определенный смысл: объекты лучше проверять после того, как они будут модифицированы. Путь следования запросов через веб-хуки доступа показан на рис. 17.2.

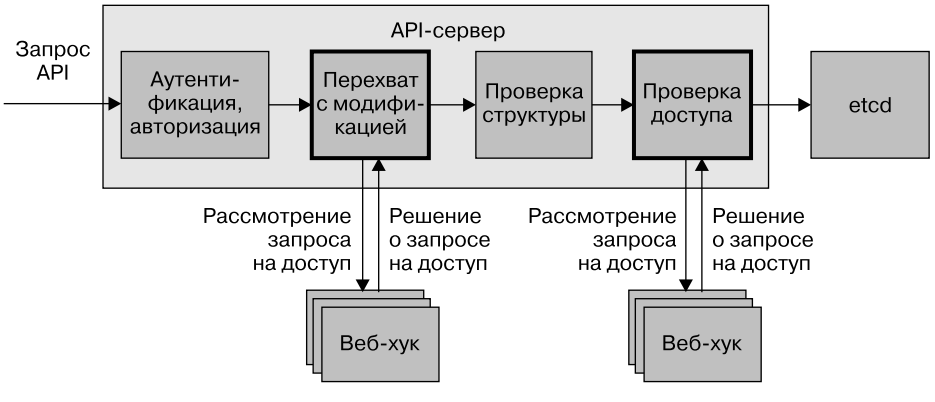


Рис. 17.2. Путь прохождения запроса API через веб-хуки доступа

- Не изменяйте одни и те же поля. С конфигурацией нескольких изменяющих веб-хуков доступа тоже могут возникнуть трудности. Вы не можете расставить их в определенном порядке, поэтому важно, чтобы они не моди-

фицировали те же поля, иначе можно получить неожиданные результаты. В целом, чтобы убедиться в соответствии итогового манифеста ресурса вашим ожиданиям, мы советуем использовать проверяющие веб-хуки доступа, поскольку они всегда выполняются вслед за изменяющими.

- `Ignore`/`Fail`. Вы, наверное, обратили внимание на поле `failurePolicy` в конфигурационных ресурсах изменяющего и проверяющего веб-хуков. Оно определяет, каким образом API-сервер должен поступить в случае, если веб-хук столкнулся с проблемами доступа или некими нераспознанными ошибками. Это поле может быть равно либо `Ignore`, либо `Fail`. Первое фактически означает, что обработка запроса будет продолжаться, тогда как второе отклоняет весь запрос. Такая логика может показаться довольно очевидной, но последствия в обоих случаях требуют внимания с вашей стороны. Если проигнорировать важный веб-хук доступа, то к ресурсу без ведома пользователя может быть не применена политика, реализующая бизнес-требования.

  Одно из потенциальных решений для защиты от подобных проблем состоит в том, чтобы сгенерировать уведомление, если в журнале API-сервера обнаружится запись о неудачном обращении к веб-хуку доступа. Значение `Fail` может иметь еще более серьезные последствия, поскольку при возникновении проблем в веб-хуке доступа все запросы будут отклоняться. Во избежание этого вы можете сделать так, чтобы веб-хуку направлялись только определенные запросы ресурсов. Старайтесь следить за тем, чтобы ни одно из ваших правил не применялось сразу ко всем ресурсам кластера.

- Если вы написали свой собственный веб-хук доступа, то помните: скорость, с которой он принимает решение и возвращает ответ, непосредственно влияет на пользовательские/системные запросы. Время ожидания всех вызовов веб-хуков доступа составляет 30 с, и по его истечении применяется политика `failurePolicy`. Даже если на принятие/отклонение запроса уходит всего несколько секунд, это может серьезно сказаться на взаимодействии пользователей с вашим кластером. Избегайте в своих веб-хуках доступа сложной логики и обращения к внешним системам, таким как базы данных.

- Ограничивайте область действия веб-хуков доступа. В манифесте веб-хука есть необязательное поле, `namespaceSelector`, позволяющее указать пространство имен, в котором он должен действовать. По умолчанию

это поле остается пустым и соответствует всему кластеру, но вы можете указать в нем метки, которые будут сопоставляться с полями `matchLabels`, принадлежащими пространствам имен. Мы советуем вам всегда использовать это поле для явного выбора подходящих пространств.

- Пространство имен `kube-system` зарезервировано во всех кластерах Kubernetes для выполнения сервисов системного уровня. Мы рекомендуем никогда не применять свои веб-хуки к ресурсам, находящимся в этом пространстве. Просто не указывайте его в поле `namespaceSelector`. То же самое относится к любым другим системным пространствам имен, которые необходимы для работы кластера.
- Ограничьте конфигурацию своих веб-хуков с помощью RBAC. Создание `MutatingWebhookConfiguration` и `ValidatingWebhookConfiguration`, разумеется, требует администраторских привилегий, и RBAC может помочь вам с ограничением прав доступа, которые получают веб-хуки. Если этого не сделать, то ваш кластер может выйти из строя или, что еще хуже, быть взломан за счет внедрения в рабочие задания ваших приложений.
- Не отправляйте чувствительные данные. Веб-хуки доступа фактически черные ящики, которые принимают на вход запросы `AdmissionRequest` и возвращают ответы `AdmissionResponse`. Пользователь не знает, как они хранят и обрабатывают его данные. Очень важно следить за тем, что именно передается веб-хукам доступа вместе с запросами. Например, объекты `Secret` и `ConfigMap` могут содержать конфиденциальную информацию и требуют четких гарантий относительно ее хранения и распространения. Передача этих ресурсов веб-хукам доступа может привести к утечке чувствительных данных, вследствие чего правила проверки и/или изменения должны иметь как можно меньшую область действия.

Авторизация

Процесс авторизации часто сводят к ответу на следующий вопрос: «Может ли данный пользователь выполнять эти действия с этими ресурсами?» В Kubernetes авторизация каждого запроса проводится после аутентификации, но перед контролем доступа. В текущем разделе мы поговорим о настройке разных модулей авторизации и поможем лучше понять, как создавать подходящие политики для вашего кластера. На рис. 17.3 показано место, которое в процессе обработки запроса занимает авторизация.

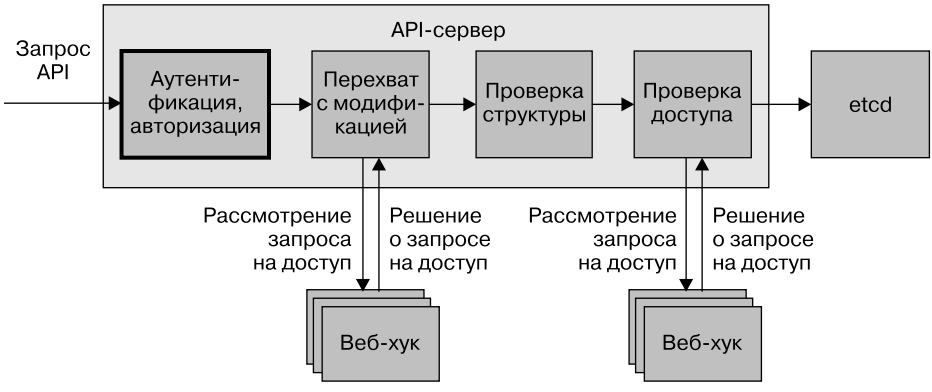


Рис. 17.3. Путь прохождения запроса API через модули авторизации

Модули авторизации

Модули авторизации отвечают за выдачу или невыдачу разрешения на доступ. Решение принимается на основе политики, которая должна быть явно задана, иначе все запросы будут по умолчанию отклоняться.

На момент выхода версии 1.15 в стандартную поставку Kubernetes входили следующие модули авторизации:

- *ABAC* (Attribute-Based Access Control — управление доступом на основе атрибутов) — позволяет конфигурировать политику авторизации с помощью локальных файлов;
- *RBAC* — позволяет конфигурировать политику авторизации с помощью API Kubernetes (см. главу 4);
- *Webhook* — делегирует авторизацию запроса удаленной конечной точке REST;
- *Node* — специальный модуль, который авторизует запросы, отправляемые утилитой kubelet.

Эти модули настраивает администратор кластера, применяя флаг API-сервера `--authorization-mode`. Их может быть несколько, и они выполняются в определенном порядке. В отличие от контроллеров доступа, запрос принимается, если его одобрил хотя бы один из модулей авторизации. Пользователь получает ошибку только в случае, если запрос был отклонен всеми модулями.

ABAC

Рассмотрим определение политики в контексте применения модуля авторизации ABAC. Следующее правило выдает пользователю Мэри (Mary) доступ на чтение к pod в пространстве имен `kube-system`:

```
apiVersion: abac.authorization.kubernetes.io/v1beta1
kind: Policy
spec:
  user: mary
  resource: pods
  readonly: true
  namespace: kube-system
```

Если Мэри сделает запрос, показанный ниже, то он будет отклонен, поскольку у Мэри нет доступа к pod в пространстве имен `demo-app`:

```
apiVersion: authorization.k8s.io/v1beta1
kind: SubjectAccessReview
spec:
  resourceAttributes:
    verb: get
    resource: pods
    namespace: demo-app
```

В этом примере используется группа API `authorization.k8s.io`, которая нам еще не встречалась. Она позволяет использовать API-сервер для авторизации внешних сервисов и включает в себя следующие API (к слову, отлично подходящие для отладки):

- `SelfSubjectAccessReview` — рассмотрение запроса на доступ для текущего пользователя;
- `SubjectAccessReview` — то же, что и `SelfSubjectAccessReview`, только для любого пользователя;
- `LocalSubjectAccessReview` — то же, что и `SubjectAccessReview`, но в рамках отдельного пространства имен;
- `SelfSubjectRulesReview` — возвращает список действий, которые пользователь может выполнять в заданном пространстве имен.

Но самое интересное в том, что обращение к этим API происходит путем создания ресурсов, как принято в Kubernetes. Вернемся к предыдущему примеру и проверим его с помощью `SelfSubjectAccessReview`. Поле `status` в выводе говорит о том, что данный запрос допускается:

```
$ cat << EOF | kubectl create -f - -o yaml
apiVersion: authorization.k8s.io/v1beta1
```

```
kind: SelfSubjectAccessReview
spec:
  resourceAttributes:
    verb: get
    resource: pods
    namespace: demo-app
EOF
apiVersion: authorization.k8s.io/v1beta1
kind: SelfSubjectAccessReview
metadata:
  creationTimestamp: null
spec:
  resourceAttributes:
    namespace: kube-system
    resource: pods
    verb: get
status:
  allowed: true
```

На самом деле утилита `kubectl`, поставляемая вместе с Kubernetes, делает этот процесс еще проще. Команда `kubectl auth can-i` обращается к тому же API, что и код из предыдущего листинга:

```
$ kubectl auth can-i get pods --namespace demo-app
yes
```

Если у вас есть администраторские права доступа, то вы можете проверить с помощью той же команды действия другого пользователя:

```
$ kubectl auth can-i get pods --namespace demo-app --as mary
yes
```

RBAC

Управление доступом на основе ролей в Kubernetes подробно рассматривается в главе 4.

Webhook

С помощью модуля webhook администратор кластера может сконфигурировать внешнюю конечную точку REST и делегировать ей процесс авторизации. Она станет находиться за пределами кластера, и к ней можно будет обращаться по URL. Конфигурация конечной точки REST хранится в файловой системе ведущего узла, а путь к ней указывается с помощью флага API-сервера `--authorization-webhook-config-file=ИМЯ_ФАЙЛА`. В результате к телу запросов, которые API-сервер направляет приложению с веб-хуком авторизации, будут прикрепляться объекты `SubjectAccessReview`, а в ответ станут возвращаться те же объекты, но с заполненным полем `status`.

Практические советы относительно авторизации

Прежде чем вносить изменения в модули авторизации, сконфигурированные в вашем кластере, обратите внимание на следующее.

- Учитывая, что политики ABAC должны храниться в файловой системе каждого ведущего узла и при этом синхронизироваться между собой, мы в целом *не рекомендуем* применять ABAC в кластерах с несколькими ведущими узлами. То же самое относится к модулю webhook, поскольку его конфигурация основана на файле и требует наличия соответствующего флага. Более того, чтобы применить измененные политики, хранящиеся в этих файлах, необходимо перезапустить API-сервер; в кластере с одним ведущим узлом вследствие такого действия фактически возникнут перебои в работе всего управляющего уровня, а в кластере с несколькими ведущими узлами появится несогласованность в конфигурации. Учитывая эти нюансы, для авторизации лучше использовать только модуль RBAC, поскольку его правила конфигурируются и хранятся внутри Kubernetes.
- Несмотря на свои богатые возможности, модули webhook могут быть крайне опасными. Через процесс авторизации проходит каждый запрос; как следствие, сбой веб-хука может иметь разрушительные последствия для кластера. В связи с этим мы в целом рекомендуем использовать внешние модули авторизации только в том случае, если вы абсолютно уверены в том, что ваш кластер сможет справиться с отказом или недоступностью сервиса, который предоставляет веб-хук.

Резюме

В данной главе мы обсудили основополагающие аспекты контроля доступа и авторизации и привели общепринятые рекомендации. Надеемся, это поможет вам выбрать наиболее подходящую конфигурацию, которая позволит адаптировать средства управления и политики, необходимые для работы вашего кластера.

ГЛАВА 18

В заключение

Главное преимущество платформы Kubernetes состоит в ее модульности и универсальности. Она позволяет разворачивать практически любые виды приложений, и все корректировки и изменения, которые необходимо внести в вашу систему, обычно под силу большинству разработчиков.

Конечно, за модульность и универсальность приходится платить несколько повышенным уровнем сложности. Понимание того, как работают API и компоненты Kubernetes, — необходимое условие для раскрытия настоящего потенциала этой платформы. Но взамен вы получаете возможность сделать процесс разработки, администрирования и развертывания своих приложений более простым и надежным.

Точно так же, чтобы эффективно применять платформу Kubernetes в реальных условиях, вы должны уметь подключать ее к большому спектру внешних систем, таких как локально размещенные базы данных и сервисы непрерывной доставки.

На страницах этой книги мы пытались передать вам реальный опыт работы над разными задачами, с которыми сталкиваются как новички, так и бывалые администраторы. Надеемся, что представленная здесь информация поможет вам стать специалистом в новой для себя сфере или просто освежить знания и посмотреть, как другие люди справляются с уже знакомыми вам проблемами. Нам хотелось бы, чтобы в процессе чтения вы приобрели навыки, которые позволят раскрыть весь потенциал Kubernetes. Спасибо за внимание, с нетерпением ждем вас в мире реальных проектов!

Об авторах

Брендан Бернс — известный инженер Microsoft Azure и соучредитель открытого проекта Kubernetes. Занимается созданием облачных приложений уже больше десяти лет.

Эдди Вильяльба работает программистом в отделе разработки коммерческого ПО компании Microsoft, занимаясь развитием открытых облачных технологий и Kubernetes. Многие пользователи могут быть благодарны за его помощь с внедрением Kubernetes в их приложения.

Дейв Штребель работает архитектором глобальных облачных систем в Microsoft Azure и занимается открытыми облачными технологиями и Kubernetes. Он принимает активное участие в открытом проекте Kubernetes, помогая с релизами новых версий и возглавляя группу SIG-Azure.

Лахлан Эвенсон — руководитель команды контейнерных вычислений в Microsoft Azure. Его практические уроки и выступления на конференциях помогли огромному количеству людей перейти на Kubernetes.

Об изображении на обложке

На обложке изображена кряква (лат. *Anas platyrhynchos*) — разновидность речной утки, которая, вместо того чтобы нырять за добычей, кормится на поверхности воды. Птицы этого семейства обитают в разных районах и имеют различные повадки; тем не менее кряквы часто скрещиваются с другими видами, давая плодовитое гибридное потомство.

Утята кряквы вылупляются уже вполне сформированными и способными самостоятельно плавать, а через 3–4 месяца начинают летать. Зрелость наступает в возрасте 14 месяцев, а средняя продолжительность жизни составляет три года.

Кряквы имеют средний размер по утиным стандартам и весят чуть больше, чем большинство речных уток. Взрослые особи достигают 58 см в длину, имеют размах крыльев 91 см и вес около 1 кг. Для утят характерно желтое и черное оперение. В возрасте примерно шести месяцев самцов и самок уже можно различить по их расцветке. Самцы имеют зеленую голову, белый «ошейник», шоколадно-коричневую грудь, серо-коричневые крылья и желтовато-оранжевый клюв. Самки кряквы, как и большинства других речных уток, окрашены в бурый цвет с темными пятнышками.

Кряквы обитают на широких просторах как в Северном, так и в Южном полушарии. Они встречаются в пресных и соленых водоемах, включая озера, реки и морские побережья. Северные кряквы зимой мигрируют на юг. Их пища очень разнообразна и может включать в себя растения, семена, корни, моллюсков, беспозвоночных и ракообразных.

Другие птицы могут откладывать свои яйца в гнездах крякв. Если кряква не заметит разницу, то будет заботиться о вылупившихся птенцах как о своих собственных.

На крякв охотится целый ряд хищников — в частности лисы и хищные птицы, такие как соколы и орлы. В воде их подстерегают сомы и щуки. В борьбе за территорию на уток, как известно, нападают вороны, лебеди и гуси. Такое

явление, как асимметричный сон, когда спит только одно полушарие мозга, а другое бодрствует, впервые было обнаружено у крякв. Это защитный механизм, который свойственен многим водоплавающим птицам.

Многие животные с обложек издательства O'Reilly находятся под угрозой исчезновения, все они важны для нашей планеты.

Титульная иллюстрация создана Хозе Марзаном на основе черно-белой гравюры из книги *The Animal World*.

Брендан Бернс, Эдди Вильяльба,
Дейв Штребель, Лахлан Эвенсон

Kubernetes: лучшие практики

Перевел с английского *А. Павлов*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*А. Сидоров*
Литературный редактор	*Н. Хлебина*
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: 01.2021. Наименование: книжная продукция. Срок годности: не ограничен.

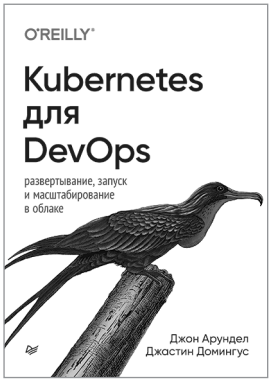
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

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 25.12.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 23,220. Тираж 700. Заказ 0000.

Джон Арундел, Джастин Домингус

KUBERNETES ДЛЯ DEVOPS: РАЗВЕРТЫВАНИЕ, ЗАПУСК И МАСШТАБИРОВАНИЕ В ОБЛАКЕ



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 над следующими приложениями.

- Начнете работу с контейнерами и Kubernetes с азов: никакого специального опыта для изучения темы не требуется.
- Запустите собственные кластеры или выберете управляемый сервис Kubernetes от Amazon, Google и др.
- Примените Kubernetes для управления жизненным циклом контейнера и расхода ресурсов.
- Оптимизируете кластеры по показателям стоимости, производительности, устойчивости, мощности и масштабируемости.
- Изучите наилучшие инструменты для разработки, тестирования и развертывания ваших приложений.
- Воспользуетесь актуальными отраслевыми практиками для обеспечения безопасности и контроля.
- Внедрите в компании принципы DevOps, чтобы команды разработчиков стали действовать более гибко, быстро и эффективно.