O'REILLY®

РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ

ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Брендан Бёрнс

ПИТЕР®

Designing Distributed Systems

Patterns and Paradigms for Scalable, Reliable Services

Brendan Burns

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Брендан Бёрнс

РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ

ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

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

2019

ББК 32.988.02-018
УДК 004.738.2
Б51

Бёрнс Б.

Б51 Распределенные системы. Паттерны проектирования. — СПб.: Питер, 2019. — 224 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-0950-0

Современный мир попросту немыслим без использования распределенных систем. Даже у простейшего мобильного приложения есть API, через который оно подключается к облачному хранилищу. Однако проектирование распределенных систем до сих пор остается искусством, а не точной наукой. Необходимость подвести под нее серьезный базис назрела давно, и, если вы хотите обрести уверенность в создании, поддержке и эксплуатации распределенных систем — начните с этой книги!

Брендан Бёрнс, авторитетнейший специалист по облачным технологиям и Kubernetes, излагает в этой небольшой работе абсолютный минимум, необходимый для правильного проектирования распределенных систем. Эта книга описывает неустаревающие паттерны проектирования распределенных систем. Она поможет вам не только создавать такие системы с нуля, но и эффективно переоборудовать уже имеющиеся.

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

ББК 32.988.02-018
УДК 004.738.2

ISBN 978-1491983645 англ.

Authorized Russian translation of the English edition of Designing Distributed Systems ISBN 9781491983645

ISBN 978-5-4461-0950-0

Краткое содержание

Оглавление

Часть I. Одноузловые паттерны проектирования

Часть II. Паттерны проектирования обслуживающих систем

Предисловие

Кому стоит прочесть эту книгу

На сегодняшний день почти каждый разработчик является создателем и/или потребителем распределенных систем. Даже относительно простые мобильные приложения опираются на облачные API, чтобы обеспечить доступность данных на любом устройстве, которым пожелает воспользоваться клиент. Будете ли вы новичком в разработке распределенных систем или закаленным в боях ветераном, паттерны и компоненты, описанные в этой книге, помогут превратить разработку таких систем из искусства в науку. Повторно используемые компоненты и паттерны проектирования распределенных систем позволят вам сосредоточиться на важных деталях вашего приложения. Это издание поможет любому разработчику более качественно, эффективно и быстро создавать распределенные системы.

Зачем я написал эту книгу

За свою карьеру разработчика программных систем — от вебпоисковиков до облачных систем — я создал множество масштабируемых, надежных распределенных систем. Каждая из

них была, по большому счету, разработана с нуля. В целом это характерно для всех распределенных приложений. Несмотря на то что зачастую многие их принципы и логика работы совпадают, шаблонные решения или повторно используемые компоненты не так-то просто применять. Это заставляло меня впустую тратить время на реализацию систем, качество которых могло быть лучше, чем оказывалось в конечном итоге.

Появившиеся недавно технологии контейнеров и их оркестраторов фундаментально изменили ландшафт разработки распределенных систем. В наше распоряжение попал объект и интерфейс, которые позволяют выражать базовые паттерны проектирования распределенных систем и компоновать контейнеризованные компоненты. Я написал эту книгу, чтобы сблизить нас, практикующих специалистов в области распределенных систем; чтобы у нас появились общий язык и общая стандартная библиотека; чтобы мы могли быстрее создавать более качественные системы.

Современный мир распределенных систем

Когда-то, много лет тому назад, люди писали программы, работавшие на той же машине, на которой к ним получали доступ пользователи. С тех пор ситуация изменилась. Теперь почти каждое приложение является *распределенной системой*, которая работает на многих машинах и к которой получает доступ множество пользователей по всему миру. Несмотря на их повсеместное распространение, проектирование и реализация таких систем — «черная магия», которой владеют лишь избранные. Но, как и все другие технологии, мир распределенных систем развивается, упорядочивается и абстрагируется. В этой книге я описываю набор обобщенных повторяемых паттернов (шаблонов),

которые делают разработку надежных распределенных систем более доступной и эффективной. Внедрение паттернов проектирования и повторно используемых компонентов освобождает разработчиков от необходимости повторно реализовывать одни и те же системы. В сэкономленное время можно сосредоточиться на разработке ключевых частей приложения.

Как ориентироваться в книге

Книга разделена на четыре части следующим образом.

- *Глава 1. Введение*. Вводит понятие распределенной системы и объясняет, каким образом паттерны проектирования и повторно используемые компоненты способствуют быстрой разработке надежных распределенных систем.
- *Часть I. Одноузловые паттерны проектирования*. В главах 2–4 обсуждаются повторно используемые паттерны и компоненты, имеющие место в рамках одного узла распределенной системы. В частности, рассматриваются паттерны Sidecar, Adapter и Ambassador.
- *Часть II. Паттерны проектирования обслуживающих систем*. В главах 8–9 рассматриваются многоузловые паттерны, применяемые в постоянно работающих обслуживающих системах, таких как веб-приложения. Обсуждаются паттерны репликации, масштабирования и выбора главного узла.
- *Часть III. Паттерны проектирования систем пакетных вычислений*. В главах 10–12 рассматриваются паттерны распределенных систем для широкомасштабной обработки данных, в том числе очереди задач, событийно-ориентированная обработка и согласованные рабочие процессы.

Если вы опытный разработчик распределенных систем, можете пропустить первые несколько глав. Тем не менее вам

стоит хотя бы пролистать их, чтобы понять, как применять паттерны проектирования и почему считается, что сама идея паттернов проектирования распределенных систем настолько важна.

Многие, вероятно, найдут полезными одноузловые паттерны, так как они являются наиболее универсальными и их проще всего использовать повторно.

В зависимости от ваших целей и от того, какие системы вы собираетесь разрабатывать, имеет смысл сосредоточиться либо на паттернах обработки больших объемов данных, либо на паттернах проектирования постоянно работающих серверов (либо и на тех и на других). Части II и III практически не зависят друг от друга, и их можно читать в любом порядке.

Если вы имеете обширный опыт разработки распределенных систем, то, возможно, посчитаете некоторые паттерны из первых глав избыточными (например, описанные в части II именование, обнаружение, распределение нагрузки). Тогда можете их просто пролистать, чтобы получить общее представление, но по пути не забудьте рассмотреть все иллюстрации!

Условные обозначения

В данной книге приняты следующие условные обозначения.

Курсив

Курсивом выделяются новые термины, слова, на которых сделан акцент.

Рубленый шрифт

Применяется для отображения URL, адресов электронной почты.

`Моноширинный шрифт`

Используется для текстов программ, а также внутри основного текста для обозначения элементов программы: имен функций, баз данных, типов данных, переменных среды, выражений и ключевых слов. Им же выделяются имена и расширения файлов.

Так обозначается совет, подсказка или общая заметка.

Онлайн-ресурсы

Хотя в этой книге описаны популярные паттерны проектирования распределенных систем, ожидается, что читатели знакомы с контейнерами и системами их оркестровки. Если же у вас недостаточно знаний об этих программных продуктах, рекомендую воспользоваться следующими ресурсами:

- https://docker.io;
- https://kubernetes.io;
- https://dcos.io.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. п.) доступны для загрузки по следующему адресу: https://github.com/brendandburns/designing-distributed-systems.

Это издание призвано помочь вам в вашей работе. В общем случае вы можете использовать поставляемые с книгой примеры кода в своих программах и документации. Нет необходимости обращаться к нам за разрешением на использование кода, за

исключением тех случаев, когда копируются его существенные фрагменты. К примеру, если вы напишете программу, использующую несколько фрагментов кода из данной книги, то вам не нужно получать разрешение. Продажа или распространение компакт-диска с исходными текстами, однако, требуют разрешения. Использование цитат и фрагментов кода из книги при ответе на вопросы не требует получения разрешения. Включение значительной части кода в документацию по вашему продукту требует разрешения.

Мы ценим, хотя и не требуем ссылки на первоисточник. Ссылка обычно включает название книги, имя автора, название издателя и номер ISBN. Например: «Распределенные системы. Паттерны проектирования. Брендан Бёрнс (O'Reilly). Авторские права защищены, Брендан Бёрнс, 2018, ISBN 978-5-4461-0950-0».

Если вам кажется, что вы выходите за рамки правомерного использования примеров кода, связывайтесь с нами по адресу permissions@oreilly.com.

Благодарности

Я хотел бы поблагодарить свою жену Робин и своих детей за все то, что они делали, чтобы я оставался здоровым и счастливым. Большое спасибо тем людям на моем жизненном пути, которые помогли мне освоить все то, о чем написано в этой книге! Отдельное спасибо моим родителям за первый Macintosh SE/30.

1 Введение

В современном мире постоянно работающих приложений и программных интерфейсов (API) к ним предъявляются такие требования, которые пару десятилетий назад предъявлялись только к небольшому количеству наиболее важных систем. Аналогичным образом наличие возможности быстрого, «вирусного» роста популярности сервиса означает, что любое приложение должно создаваться с расчетом на почти мгновенное масштабирование в ответ на увеличивающийся пользовательский спрос. Эти ограничения и требования означают, что почти каждое разрабатываемое приложение, будь то мобильная клиентская программа или сервис обработки платежей, должно быть распределенной системой.

Но строить распределенные системы непросто. Как правило, это единичные заказные системы. В этом смысле разработка распределенных систем поразительно похожа на разработку программного обеспечения (ПО) в тот период, когда еще не существовало современных объектно-ориентированных языков программирования. К счастью, как и в случае с созданием

объектно-ориентированных языков, имел место технический прогресс, существенно снизивший трудоемкость построения распределенных систем. В данном случае он связан с растущей популярностью контейнеров и инструментов оркестрирования.

Подобно объектам в объектно-ориентированном программировании, контейнеризованные строительные блоки — основа разработки повторно используемых компонентов и паттернов проектирования. Они существенно упрощают создание надежных распределенных систем и делают его более доступным для начинающих разработчиков. Далее будет кратко описана история разработок, приведших к современному состоянию отрасли.

Краткая история разработки систем

Первое время машины создавались для какой-то одной цели: расчета артиллерийских таблиц, прогнозирования приливов и отливов, взлома шифров и других точных, сложных, но рутинных математических вычислений. Спустя годы специализированные машины превратились в программируемые компьютеры общего назначения. Те со временем перешли от выполнения одной программы на одной машине к параллельному выполнению многих программ на одной машине с помощью операционных систем с разделением времени. Эти машины все еще были разъединены друг с другом.

Постепенно машины стали объединяться в сети, в результате чего появились клиент-серверные архитектуры. Относительно маломощные компьютеры на рабочих местах получили доступ к вычислительным ресурсам мощных мейнфреймов, находящихся в других помещениях или даже зданиях. И хотя такой вид клиент-серверного программирования был несколько сложнее написания программы для одного компьютера, он все

еще был относительно прост для понимания. Клиент (-ы) делал (-и) запросы, а сервер (-ы) их обслуживал (-и).

Рост Интернета и появление в начале 2000-х крупных центров обработки данных (ЦОД), состоящих из тысяч относительно недорогих массово производимых компьютеров, которые объединялись в сеть, привели к широкому распространению *распределенных систем*. В отличие от клиент-серверных архитектур распределенные приложения состоят либо из нескольких разных приложений, либо из нескольких копий одного приложения, работающих на разных машинах. Взаимодействуя, они реализуют некоторый сервис, например веб-поисковик или систему розничных продаж.

В силу своего распределенного характера такие системы при грамотной их структуризации более надежны по определению. А при грамотно спроектированной архитектуре системы масштабируемой становится и ее команда разработчиков. К сожалению, за эти преимущества приходится платить. Распределенные системы существенно сложнее в проектировании, построении и отладке. При построении надежной распределенной системы к инженерно-техническим навыкам специалистов предъявляются существенно более высокие требования, чем при построении локальных приложений. Так или иначе, потребность в надежных распределенных системах продолжает расти. Следовательно, возникает необходимость в соответствующих инструментах, паттернах и практиках их построения.

К счастью, современные технологии упрощают разработку распределенных систем. В последние годы контейнеры, их образы и оркестраторы стали популярными в силу того, что являются неотъемлемыми составными частями надежных распределенных систем. Взяв за основу контейнеры и оркестраторы контейнеров, мы можем создать набор повторно используемых

компонентов и паттернов проектирования. Такие паттерны и компоненты составляют инструментарий, необходимый для разработки более эффективных надежных систем.

Краткая история паттернов проектирования в разработке ПО

Чтобы лучше понять, как паттерны, практики и повторно используемые компоненты изменили разработку систем, имеет смысл взглянуть на то, как подобные трансформации происходили в прошлом.

Формализация алгоритмического программирования

Люди писали программы задолго до опубликования Дональдом Кнутом сборника «Искусство программирования»[1]. Тем не менее это событие стало важной вехой в развитии информатики. В частности, описанные в книгах Кнута алгоритмы не ориентированы на какой-либо компьютер, а предназначены для обучения читателя алгоритмическому мышлению. Эти алгоритмы могут быть адаптированы к конкретной компьютерной архитектуре или к конкретной задаче, решаемой читателем. Такая формализация была важна не только потому, что предоставляла разработчикам общий инструментарий для написания программ, но и потому, что демонстрировала существование универсальных идей, которые можно применять в разнообразных контекстах. Понимание алгоритмов имеет ценность само по себе, безотносительно к какой-либо решаемой с их помощью задаче.

---

[1] *Кнут Д.* Искусство программирования. В 4 т. — М.: Вильямс, 2017.

Паттерны в объектно-ориентированном программировании

Если появление книг Кнута стало важной вехой в теории компьютерного программирования, то алгоритмы — ключевой составляющей его развития. Однако по мере роста сложности компьютерных программ и увеличения численного состава разрабатывающих их команд с единиц до сотен и тысяч стало ясно, что языков процедурного программирования и алгоритмов уже недостаточно для решения насущных задач. Эти изменения привели к появлению и развитию объектно-ориентированных языков программирования, которые уравняли в правах с алгоритмами данные, повторное использование и расширяемость.

В ответ на эти изменения в компьютерном программировании изменениям подверглись также паттерны и практики программирования. В начале и середине 1990-х годов произошел взрывной рост количества книг об объектно-ориентированном программировании. Наиболее известна из них книга «банды четырех» «Приемы объектно-ориентированного проектирования»[1].

«Паттерны проектирования» привнесли в работу программистов инфраструктуру и общий язык. В этой книге описывается набор интерфейсных паттернов, которые можно использовать в различных контекстах. Благодаря развитию объектно-ориентированного программирования в целом и интерфейсов в частности появилась возможность реализовать такие паттерны в виде повторно используемых библиотек. Эти библиотеки можно написать единожды и затем многократно использовать, экономя тем самым время и повышая надежность.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования: паттерны проектирования. — СПб.: Питер, 2015.

Расцвет программного обеспечения с открытым исходным кодом

Делиться исходным кодом с другими разработчиками было принято со времен появления вычислительной техники. Формальные организации в поддержку свободного программного обеспечения существовали с середины 1980-х. Но именно в конце 1990-х — начале 2000-х резко возросло количество разработчиков и потребителей программного обеспечения с открытым исходным кодом. Хотя движение open source лишь относительно связано с разработкой паттернов проектирования распределенных систем, его заслуга состоит в том, что именно сообщества open source показали миру: создание программного обеспечения в целом и распределенных систем в частности — труд коллективный.

Здесь важно отметить, что все технологии контейнеров, лежащие в основе паттернов проектирования из этой книги, разрабатывались и выпускались именно как программное обеспечение с открытым исходным кодом. Ценность паттернов для документирования и усовершенствования практик разработки распределенных программных систем становится наиболее очевидной именно с точки зрения коллективной разработки.

Что такое паттерн проектирования распределенной системы? Написано множество инструкций по установке конкретных распределенных систем (например, баз данных NoSQL). Кроме того, у каждого набора систем (например, стека MEAN[1]) есть свои правила установки. Но когда я говорю о паттернах, я имею в виду обобщенные схемы организации распределенных систем, не зависящие от конкретных технологий или приложений.

[1] Типовой набор средств разработки, включающий инструменты MongoDB, Express.js, Angular, Node.js. — *Здесь и далее примеч. пер.*

> Цель паттерна — предоставить общие предложения по архитектуре системы, задать ее ориентировочную структуру. Надеюсь, что эти паттерны направят ход ваших мыслей в верную сторону и окажутся применимы в широком спектре приложений и программных сред.

Ценность паттернов, практик и компонентов

Прежде чем тратить ценное время на чтение книги о наборе паттернов, которые, с моих слов, усовершенствуют ваши подходы к работе, научат вас новым приемам разработки и — давайте посмотрим правде в глаза — изменят вашу жизнь, имеет смысл спросить: «А зачем?» Что такого есть в паттернах и методиках разработки, что может поменять подход к проектированию и компоновке программного обеспечения? В этом разделе я объясню, почему считаю их важными. Надеюсь, мои доводы убедят вас прочесть книгу полностью.

Стоя на плечах гигантов[1]

Начнем с того, что паттерны проектирования распределенных систем позволяют, образно говоря, стоять на плечах гигантов. Задачи, которые мы решаем, или системы, которые мы создаем, нечасто становятся действительно уникальными. В конечном итоге собранные воедино компоненты и общая бизнес-модель, которую позволяет организовать разрабатываемое программное обеспечение, являются чем-то новым. Но то, как система построена, и те проблемы, с которыми она сталкивается в своем стремлении быть надежной и масштабируемой, отнюдь не новы.

---

[1] Отсылка к высказыванию, приписываемому Ньютону: «Если я видел дальше других, то потому, что стоял на плечах гигантов».

В этом, стало быть, состоит первая ценность паттернов: они позволяют учиться на чужих ошибках. Возможно, вы никогда раньше не разрабатывали распределенные системы. Возможно, вы никогда раньше не разрабатывали определенный вид распределенных систем. Вместо того чтобы надеяться на опыт вашего коллеги в этой области или учиться на ошибках, которые совершали другие, вы можете обратиться за помощью к паттернам.

Изучение паттернов проектирования распределенных систем — то же самое, что изучение любых других передовых практик компьютерного программирования. Оно ускоряет разработку программного обеспечения, не требуя наличия непосредственного опыта разработки систем, исправления ошибок и набивания собственных шишек.

Общий язык обсуждения подходов к разработке

Возможность учиться, а также лучше и быстрее понимать разработку распределенных систем — лишь часть преимуществ от наличия общего набора паттернов проектирования. Паттерны ценны даже для опытных разработчиков распределенных систем, уже хорошо их понимающих. Паттерны предоставляют общий словарь, позволяющий нам быстро понимать друг друга. Понимание — основа обмена знаниями и дальнейшего обучения.

Для того чтобы было понятнее, представим, что мы оба используем один и тот же инструмент для постройки дома. Я называю его «сепулька», а вы называете его «бутявка». Как долго мы будем спорить о преимуществах «сепулек» над «бутявками» или пытаться объяснить различие в их свойствах, пока не придем к тому, что это одно и то же? Только когда мы придем к тому,

что «сепульки» и «бутявки» — одно и то же, мы сможем учиться на опыте друг друга.

При отсутствии общего словаря много времени тратится либо на споры в поисках «насильственного согласия», либо на объяснение понятий, которые остальные понимают, но называют по-другому. Следовательно, ценность паттернов состоит еще и в том, что они обеспечивают наличие общего набора понятий и их определений, позволяющего не тратить время на дискуссии об именах, а перейти к обсуждению деталей реализации основных идей.

За то короткое время, что я работал над технологией контейнеров, я убедился в этом. На тот момент идея контейнеров-прицепов (будут описаны в главе 2) прочно укрепилась в сообществе «контейнерщиков». Благодаря этому не было необходимости тратить время на разъяснение того, что значит быть контейнером-прицепом, а вместо этого можно было перейти к обсуждению того, как использовать этот паттерн для решения конкретной задачи. «А вот если мы здесь используем паттерн Sidecar...» — «Ага, кажется, я знаю, какой контейнер отлично подойдет для этой задачи». Этот пример подводит нас к третьему показателю ценности паттернов проектирования — возможности создания повторно используемых компонентов.

Общие повторно используемые компоненты

Паттерны позволяют учиться на чужом опыте и предоставляют общий язык для обсуждения тонкостей построения систем, но, помимо этого, они дают программисту еще один инструмент — возможность выявлять общие компоненты, которые достаточно реализовать однократно.

Если бы мы писали весь необходимый программный код самостоятельно, то мы никогда бы ничего не доделали. Более того, у нас едва бы получалось начать. Каждая созданная или создаваемая на сегодня система является результатом тысяч, а то и сотен тысяч человеко-лет работы. Код операционных систем, драйверов принтеров, распределенных баз данных, исполнительных сред контейнеров и их оркестраторов — все, что мы сегодня строим, создается на основе совместно используемых библиотек и повторно используемых компонентов.

Паттерны — основа формирования и развития таких компонентов. Формализация алгоритмов привела к созданию повторно используемых реализаций сортировки и других канонических алгоритмов. Благодаря выявлению интерфейсных паттернов проектирования появился целый ряд объектно-ориентированных библиотек, их реализующих.

Выявление базовых паттернов проектирования распределенных систем позволяет создавать разделяемые общие компоненты таких систем. Реализация этих паттернов в виде контейнеров с HTTP-интерфейсом дает возможность использовать их в различных языках программирования. И конечно же, разработка, ориентированная на построение повторно применяемых компонентов, позволяет улучшать качество каждого из них, поскольку в используемом многими людьми коде более высока вероятность обнаружения ошибок и недостатков.

Резюме

Распределенные системы необходимы для того, чтобы обеспечить уровень надежности, гибкости и масштабируемости, ожидаемый от современных компьютерных программ.

Проектирование распределенных систем пока остается «черной магией» для посвященных, а не наукой, доступной непрофессионалу. Выявление общих шаблонов и практик упорядочило и усовершенствовало подходы к алгоритмическому и объектно-ориентированному программированию. Эта книга призвана сделать то же для распределенных систем. Поехали!

Часть I
Одноузловые паттерны проектирования

В этой книге описываются распределенные системы — приложения, состоящие из множества компонентов, работающих на множестве машин. В первой части речь пойдет о паттернах, локализованных в рамках одного узла. Мотивация этого проста. Контейнеры — основной строительный элемент паттернов, рассматриваемых в данной книге, но в конечном итоге именно группа контейнеров, локализованная на одной машине, представляет собой базовый элемент паттернов проектирования распределенных систем.

Мотивация

Нам понятно, почему возникает потребность разбить распределенное приложение на части, работающие на разных машинах. Но не так понятно, почему необходимо делить на контейнеры компонент приложения, работающий на одной машине. Чтобы разобраться в мотивации такой группировки контейнеров, стоит сначала понять цели, стоящие за контейнеризацией как таковой. В общем случае цель контейнера — установить ограничение на определенный ресурс (например, приложению нужно два процессорных ядра и 8 Гбайт оперативной памяти). Такой лимит может также привести к разделению сфер ответственности команд разработчиков (например, конкретная команда отвечает за определенный образ). Наконец, это может способствовать разделению обязанностей между частями кода (конкретный образ отвечает за конкретную функциональность).

Все эти причины побуждают делить приложение на группу контейнеров даже в пределах одной машины. Сначала рассмо-

трим изоляцию ресурсов. Ваше приложение может состоять из двух компонентов — сервера приложений, взаимодействующего с пользователем, и фонового загрузчика конфигурационных файлов. Очевидно, что в первую очередь требуется минимизировать задержку обработки пользовательских запросов, поэтому приложение, взаимодействующее с пользователем, должно иметь достаточно ресурсов, чтобы обеспечить максимальную отзывчивость. В то же время загрузчик конфигурационных файлов обычно нетребователен к времени отклика. Если он будет испытывать небольшую задержку в период максимального количества пользовательских запросов, система будет в порядке. Более того, фоновый загрузчик конфигурационных файлов не должен влиять на качество обслуживания конечных пользователей.

Исходя из всех этих причин, необходимо поместить сервис, который непосредственно взаимодействует с пользователем, и фоновый загрузчик в отдельные контейнеры. Так можно будет закрепить за ними разный объем памяти и вычислительных ресурсов, что, к примеру, позволит фоновому загрузчику по возможности «забирать» процессорное время у пользовательского сервиса в те периоды, когда он слабо нагружен. Кроме того, раздельное назначение вычислительных ресурсов двум контейнерам позволит сделать так, чтобы фоновый загрузчик завершался раньше пользовательского сервиса в случае их конфликта за ресурсы, вызванного утечкой или избыточным распределением памяти.

Кроме изоляции ресурсов, существует множество причин разделять одноузловое приложение на несколько контейнеров. Рассмотрим масштабирование команды. Есть достаточно оснований верить тому, что идеальное количество человек в команде — от шести до восьми. Чтобы так структурировать команды и при этом создавать системы значительного размера, членам каждой команды необходимо давать небольшой, четко ограниченный

участок работ, за который они несли бы ответственность. Нередко отдельные компоненты (при условии, что они правильно выделены) оказываются повторно применяемыми модулями, которыми могут воспользоваться разные команды.

Рассмотрим, к примеру, задачу синхронизации локальной файловой системы с удаленным Git-репозиторием исходных текстов. Если вы сделаете такой инструмент синхронизации отдельным контейнером, то сможете использовать его из PHP, HTML, JavaScript, Python и других веб-ориентированных языков и сред. Если же выделить каждую среду в отдельный контейнер, где, скажем, интерпретатор Python и Git-синхронизатор будут неразрывно связаны, то повторное использование такого модуля и, как следствие, существование соответствующей небольшой команды разработчиков, ответственной за этот модуль, станут невозможными.

Наконец, даже если ваше приложение невелико и за все контейнеры ответственна одна команда, имейте в виду, что разделение обязанностей способствует грамотному восприятию, тестированию, обновлению и развертыванию вашего приложения. Небольшие приложения с четко очерченными границами проще для понимания и менее жестко привязаны к другим системам. Это значит, что, к примеру, вы можете развернуть контейнер с Git-синхронизатором, не разворачивая заново сервер приложений. Благодаря этому сужается круг зависимостей и уменьшается масштаб развертывания в целом. Это, в свою очередь, обеспечивает более надежный процесс развертывания и отката на предыдущую версию, что увеличивает скорость и гибкость развертывания.

Резюме

Надеюсь, приведенные примеры натолкнули вас на мысль о декомпозиции своих приложений на несколько контейнеров, даже если они работают в рамках одного узла. В последующих главах

будут описаны паттерны, которые помогут вам сориентироваться в построении модульных групп контейнеров. В отличие от многоузловых распределенных паттернов эти паттерны подразумевают тесную связь между всеми контейнерами. В частности, они предполагают, что исполнение контейнеров в паттерне может быть надежно распланировано в рамках одной машины. Они также подразумевают, что все контейнеры в рамках паттерна могут при необходимости совместно использовать тома или части файловых систем, а также иные ключевые ресурсы, например сетевые пространства имен и общую память. Такая тесно связанная группа в Kubernetes[1] называется *подом (pod)*, но сама идея в общем случае применима к разным оркестраторам контейнеров, хотя некоторые из них могут поддерживать ее более естественным образом, нежели другие.

[1] Kubernetes — система с открытым исходным кодом для автоматизации развертывания, масштабирования и управления контейнеризованными приложениями.

2 Паттерн Sidecar

Sidecar — это одноузловой паттерн, состоящий из двух контейнеров. Первый из них — *контейнер приложения*. Он содержит основную логику программы. Без этого контейнера приложения бы не существовало. Вдобавок к контейнеру приложения предусмотрен еще *«прицепной» (sidecar) контейнер*. Роль прицепа — дополнить и улучшить контейнер приложения, часто таким образом, чтобы приложение не знало о его существовании. В простейшей форме контейнер-прицеп можно использовать, чтобы добавить функциональности контейнеру, который было бы сложно улучшить иным способом.

Исполнение контейнера-прицепа совместно с основным контейнером планируется посредством атомарной *группы контейнеров*, такой как под (pod) в Kubernetes. Контейнер-прицеп и контейнер приложения совместно используют не только процессорные, но и другие ресурсы — части файловой системы, имя хоста, сетевые (и другие) пространства имен. Обобщенная схема паттерна Sidecar приведена на рис. 2.1.

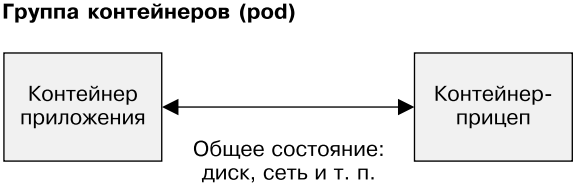

Рис. 2.1. Обобщенный паттерн Sidecar

Пример реализации паттерна Sidecar. Добавление возможности HTTPS-соединения к унаследованному сервису

Рассмотрим, к примеру, унаследованный веб-сервис. Много лет назад, когда он создавался, безопасность внутренней сети не имела такого высокого приоритета для компании, как сейчас, а значит, запросы пользователей обслуживались по незашифрованному протоколу HTTP, а не HTTPS. В связи с последними инцидентами в системе безопасности руководство компании обязало разработчиков использовать протокол HTTPS на всех сайтах компании. Солью на раны команды разработчиков, которой поручено модернизировать этот сервис, стало то, что его исходные тексты собирались на старой версии сборочной системы, которая уже не функционирует. Контейнеризовать HTTP-приложение достаточно просто — двоичный исполняемый файл может работать в старом дистрибутиве Linux поверх более современного ядра, функционирующего на оркестрационном сервере, принадлежащем команде. Но добавить HTTPS к этому приложению — задача куда более сложная.

Команде нужно принять решение — или воскресить старую сборочную систему, или портировать исходный код приложения на новую. Один из разработчиков предлагает использовать паттерн Sidecar для менее радикального разрешения этой ситуации.

Применение паттерна Sidecar в данной ситуации самоочевидно. Унаследованный веб-сервис настроен на обслуживание запросов исключительно с локального компьютера (127.0.0.1), а это значит, что к нему могут получить доступ только те сервисы, которые используют общий с ним сетевой адаптер (то есть запущены на одном компьютере). Обычно это непрактично, поскольку означает, что к такому сервису никто не сможет получить доступ. При использовании паттерна Sidecar к контейнеру с приложением добавляется контейнер-прицеп с веб-сервером nginx. Nginx-контейнер находится в том же пространстве имен, что и унаследованное веб-приложение, поэтому ему доступен сервис, работающий на localhost. В то же время nginx позволяет обслуживать HTTPS-трафик, приходящий с внешнего адреса группы контейнеров, и проксировать его унаследованному веб-приложению (рис. 2.2). Поскольку незашифрованный трафик проходит только через локальный петлевой интерфейс внутри группы контейнеров, уровень безопасности данных теперь удовлетворяет службу сетевой безопасности компании. Таким образом, команда модернизировала унаследованное приложение, не задаваясь проблемой его пересборки с целью поддержки HTTPS.

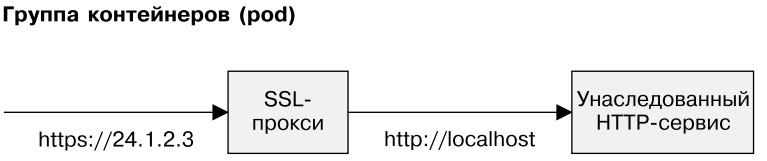

Рис. 2.2. HTTPS-прицеп

Динамическая конфигурация с помощью паттерна Sidecar

Простое проксирование трафика в уже существующее приложение не единственный случай применимости паттерна Sidecar.

Другой расхожий пример — синхронизация конфигурации. Многие приложения используют конфигурационные файлы для настройки параметров. Они могут быть простыми текстовыми файлами либо иметь более жесткую структуру, как у форматов XML, JSON или YAML. Многие приложения написаны с учетом того, что такой файл в системе существует и что они смогут считать из него свою конфигурацию. Однако в изначально облачной среде полезнее использовать API для обновления конфигурации. Это позволяет динамически разворачивать конфигурацию посредством API, а не заходить вручную на каждый сервер и редактировать конфигурационные файлы императивными командами. Желание задействовать такой API продиктовано как легкостью использования, так и возможностью автоматизации, например отката, что делает конфигурацию и реконфигурацию сервера безопаснее и проще.

По аналогии с примером про HTTPS новые приложения можно писать так, чтобы их конфигурация была динамической и ее можно было получать с помощью облачных API-запросов. Адаптация же существующего приложения и его обновление могут оказаться более сложной задачей. К счастью, паттерн Sidecar можно также использовать для расширения функциональности приложения новыми возможностями, не изменяя при этом самого приложения. В реализации паттерна Sidecar, приведенной на рис. 2.3, также показано два контейнера — контейнер, предоставляющий услуги приложения, и контейнер с менеджером конфигурации. Два контейнера объединены в под, в котором они совместно используют каталог. В этом совместно используемом каталоге и находится конфигурационный файл.

Унаследованное приложение при запуске предсказуемо загружает конфигурацию из файла в файловой системе. Менеджер конфигурации при запуске считывает данные конфигурационного

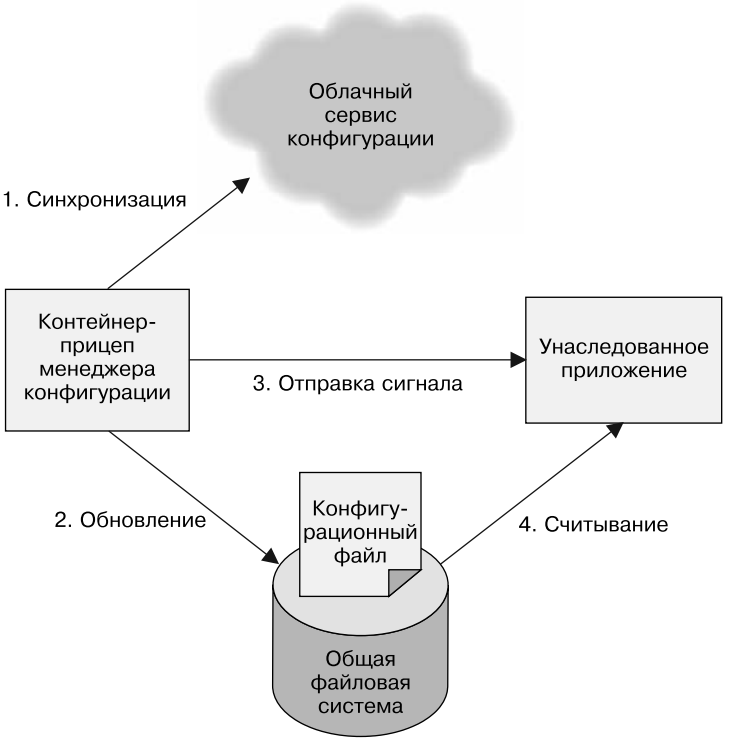

Рис. 2.3. Пример использования паттерна Sidecar для динамического управления конфигурацией

API и ищет различия между файловой системой и сохраненной посредством API конфигурацией. При обнаружении различий менеджер конфигурации загружает новые настройки в конфигурационный файл локальной файловой системы и уведомляет унаследованное приложение, что ему необходимо повторно считать конфигурацию.

Механизм такого уведомления различается от приложения к приложению. Одни приложения следят за изменением конфигурационного файла, другие ожидают сигнала SIGHUP.

В крайнем случае менеджер конфигурации может завершить приложение, отправив ему сигнал SIGKILL. После остановки приложения оркестратор перезапустит контейнер унаследованного приложения, и в этот момент оно загрузит новую конфигурацию. Как и в случае с добавлением HTTPS в существующее приложение, этот пример иллюстрирует, как паттерн Sidecar может помочь адаптировать существующие приложения к требованиям облачной среды.

Модульные контейнеры приложений

Простительно думать, что единственная причина существования паттерна Sidecar — необходимость адаптации устаревших приложений без изменения их исходных текстов. И хотя это популярный вариант использования данного паттерна, существует множество других причин проектировать приложения с его помощью. Одно из основных преимуществ применения паттерна Sidecar — модульность и повторное использование контейнеров-прицепов. Для развертывания любого «боевого» приложения, от которого ожидается высокий уровень надежности, требуется функционал, касающийся отладки и управления приложением, например считывание ресурсов, потребляемых приложениями внутри контейнера, по аналогии с утилитой командной строки top.

Одним из подходов к такой интроспекции может послужить требование реализации в каждом приложении HTTP-интерфейса `/topz`, предоставляющего срез использования ресурсов. Чтобы упростить задачу, вы можете реализовать его в виде языкозависимой надстройки, которую каждый разработчик сможет добавлять к своему приложению. В этом случае разработчик будет вынужден ее добавить, а организации придется реализовать ее для всех языков, для которых необходима ее поддержка. Такой подход при недостаточно строгой реализации почти наверняка

приведет к расхождениям между языками и отсутствию поддержки этой функциональности в новых языках.

Функциональность topz можно развернуть в виде контейнера-прицепа, работающего в том же пространстве идентификаторов процессов, что и контейнер приложения. Topz-контейнер осуществляет интроспекцию всех запущенных процессов и предоставляет единообразный пользовательский интерфейс. Кроме того, можно задействовать оркестратор для автоматического развертывания такого контейнера вместе со всеми приложениями, чтобы для каждого приложения, работающего в вашей инфраструктуре, был доступен одинаковый набор инструментов.

Любой технический выбор подразумевает некоторый компромисс между применением модульных контейнерных паттернов и внесением собственного кода в приложение. Библиотечно-ориентированный подход всегда будет несколько менее адаптирован под особенности вашего приложения. Подобная реализация может оказаться менее эффективной в плане размера или производительности; API могут потребовать некоторой адаптации для использования в вашей среде. Я бы сравнил такой компромисс с компромиссом между покупкой готовой одежды и заказом ее у модельера. Заказная одежда вам подойдет лучше, но на ее производство понадобится больше времени и она будет стоить вам дороже. Как и с вещами, когда дело касается программирования, многим из нас имеет смысл покупать решения общего назначения. Если ваше приложение требует максимальной производительности, то, конечно же, всегда можно прибегнуть к самодельному решению.

Практикум. Развертывание контейнера topz

Чтобы увидеть контейнер-прицеп в действии, сначала необходимо создать еще один контейнер, который послужит кон-

тейнером приложения. Возьмите существующее приложение и разверните его с помощью Docker:

```
$ docker run -d <образ-приложения>
<хеш-сумма-контейнера>
```

После запуска данного образа вы получите идентификатор конкретного контейнера. Он будет выглядеть как-то так: `cccf82b85000`. Если идентификатор контейнера вам неизвестен, вы всегда можете его просмотреть с помощью команды `docker ps`, которая покажет все запущенные в данный момент контейнеры.

Допустим, вы поместили это значение в переменную среды `APP_ID`. Теперь можете запустить контейнер topz в том же пространстве идентификаторов процессов с помощью такой команды:

```
$ docker run --pid=container:${APP_ID} \
    -  p 8080:8080 \
    brendanburns/topz:db0fa58
    /server -addr 0.0.0.0:8080
```

Это запустит контейнер-прицеп topz в том же самом пространстве идентификаторов процессов контейнера приложений. Заметьте, что вам, возможно, придется поменять порт, используемый прицепом, если ваш контейнер с приложением также принимает входящие запросы на порт 8080. При запущенном контейнере-прицепе вы можете обратиться к адресу http://localhost:8080/topz, чтобы получить полный срез информации о процессах, работающих внутри контейнера приложения и используемых ими ресурсах.

Вы можете применять контейнеры-прицепы совместно с любыми другими контейнерами, чтобы без труда увидеть в веб-интерфейсе, как контейнер использует ресурсы хоста.

Создание простейшего PaaS-сервиса на основе паттерна Sidecar

Паттерн Sidecar может использоваться не только для адаптирования и мониторинга. Его также можно задействовать для реализации всей логики приложения с применением упрощенного модульного подхода. Представьте себе, к примеру, простой PaaS-сервис, построенный вокруг рабочего процесса в Git-репозитории. Когда вы развернете этот сервис, вы сможете разворачивать код на рабочие серверы путем загрузки его в Git-репозиторий. Рассмотрим, как с помощью паттерна Sidecar можно простейшим образом реализовать такой PaaS.

Как было сказано ранее, в паттерне Sidecar два контейнера — основной контейнер приложения и контейнер-прицеп. В простом PaaS-приложении основной контейнер представляет собой Node.js-сервер, реализующий веб-сервис. Node.js-сервер настроен таким образом, что автоматически перезапускается при обновлении файлов. Это реализовано с помощью инструмента nodemon (https://nodemon.io/).

Контейнер-прицеп использует общую с основным контейнером приложения файловую систему и выполняет простой цикл, синхронизирующий ее с Git-репозиторием:

```
#!/bin/bash

while true; do
  git pull
  sleep 10
done
```

Безусловно, этот скрипт мог быть гораздо сложнее. Он намеренно упрощен, чтобы его было проще читать.

Node.js-приложение и прицеп с Git-синхронизатором, реализующие наш простейший PaaS-сервис, развертываются и исполня-

ются совместно на одном узле (рис. 2.4). После развертывания прицеп будет автоматически обновлять файлы в контейнере приложения по мере их загрузки в Git-репозиторий.

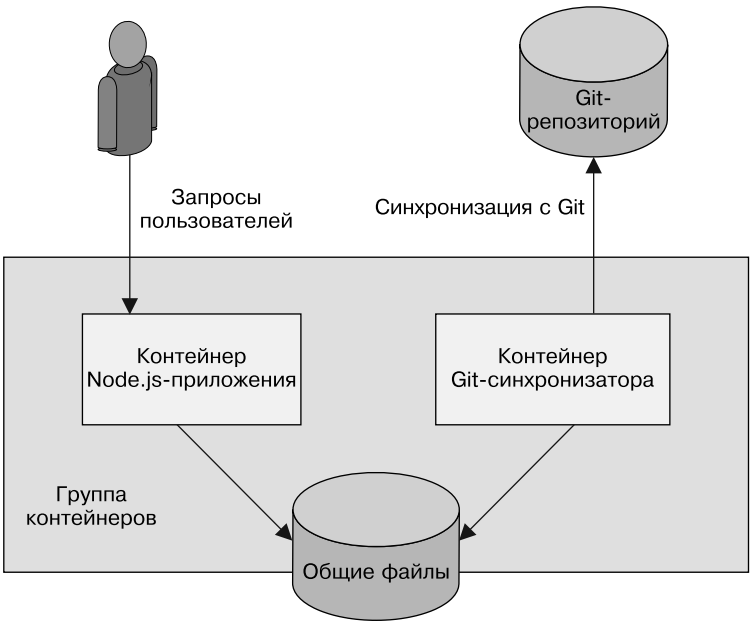

Рис. 2.4. Простейший PaaS-сервис на базе паттерна Sidecar

Разработка модульных и повторно используемых реализаций паттерна Sidecar

Во всех приведенных в данной главе примерах реализации паттерна Sidecar одной из важнейших целей было получение модульного, повторно используемого артефакта. Реализация паттерна Sidecar будет наиболее эффективной, если ее можно будет применять во множестве приложений и во множестве

сценариев развертывания. Обеспечивая модульность и возможность повторного использования, реализации этого паттерна позволяют значительно ускорить разработку вашего приложения.

Модульность и возможность повторного применения, как и достижение модульности в разработке высококлассного программного обеспечения, требуют сосредоточенности и дисциплины. В частности, необходимо сконцентрироваться на таких трех составляющих, как:

- параметризация контейнеров;
- создание программного интерфейса контейнеров;
- документирование работы контейнера.

Параметризованные контейнеры

Параметризация контейнеров — важнейший показатель достижения модульности и повторного использования контейнеров, независимо от того, реализуют они паттерн Sidecar или нет.

Что я понимаю под параметризацией? Представьте, будто контейнер — это функция в программе. Сколько у нее параметров? Каждый параметр представляет собой входные данные, которые помогают подстроить обобщенный контейнер под конкретную ситуацию. Рассмотрим, к примеру, контейнер-прицеп с SSL, который мы развернули ранее. Чтобы быть полезным в общем случае, он должен иметь по меньшей мере два параметра: имя сертификата, обеспечивающего функциональность SSL, и порт унаследованного сервера приложений, запущенного на локальной машине. Без этих параметров трудно сказать, что он будет полезен для широкого спектра приложений. Похожие параметры есть во всех контейнерах-прицепах, рассмотренных в данной главе.

Теперь, когда мы знаем, какие параметры предоставить, остается вопрос: как их, собственно, предоставить и использовать их значения в рамках контейнера. Параметры контейнеру можно передавать двумя способами — через переменные среды или через командную строку. И хотя оба они допустимы, в общем случае я предпочитаю передавать параметры с помощью переменных среды. Пример передачи параметра контейнеру:

```
docker run -e=PORT=<порт> -d <образ>
```

Передача значений в контейнер — только половина успеха. Другая половина — собственно использование значений переменных внутри контейнера. Обычно это реализуется в рамках сценариев оболочки. Такой сценарий подгружает переменные среды, переданные контейнеру-прицепу, и на их основе либо корректирует конфигурационные файлы, либо параметризует базовое приложение.

К примеру, можно передать путь к сертификату и порт приложения в виде переменных среды следующим образом:

```
docker run -e=PROXY_PORT=8080 \
  -e=CERTIFICATE_PATH=/путь/к/сертификату.crt ...
```

Сценарий в контейнере воспользуется значениями этих переменных для формирования конфигурационного файла `nginx.conf`, который укажет серверу, где искать файл сертификата и куда переадресовывать запросы.

Определение API всех контейнеров

Если учитывать, что вы параметризуете свои контейнеры, будет очевидно, что каждый из них определяет некую «функцию», которая вызывается при запуске контейнера. Эта функция является частью API, определяемого вашим контейнером, но у него есть и другие составляющие, включая вызовы к внешним по отношению

к контейнеру сервисам и предоставляемые контейнером API-услуги, доступные по HTTP или любым другим способом.

Думая о модульности и повторном использовании контейнеров, важно понимать, что программный интерфейс (API) контейнера определяется всеми его аспектами взаимодействия с внешней средой. Как и в среде микросервисов, *микроконтейнеры* рассчитывают на наличие некоторого программного интерфейса, который бы четко разделил основное приложение и контейнер-прицеп. Кроме того, наличие API гарантирует, что все потребители контейнера-прицепа будут работать корректно даже после выхода последующих его версий. В то же время наличие четкого API у контейнера-прицепа позволяет его создателю более эффективно работать, поскольку в этом случае у него есть четкое определение услуг, предоставляемых контейнером (а желательно и юнит-тестов для них).

Чтобы понять, насколько важно уделять внимание API контейнера, рассмотрим упомянутый ранее контейнер-прицеп для управления конфигурацией. Для него мог бы оказаться полезным параметр `UPDATE_FREQUENCY`, задающий частоту, с которой необходимо синхронизировать конфигурацию с файловой системой. Очевидно, что если название параметра потом поменяется на, скажем, `UPDATE_PERIOD`, то это уже будет нарушением интерфейса контейнера и воспрепятствует его корректному применению другими пользователями.

Такой пример, конечно же, очевиден, но нарушить интерфейс контейнера можно гораздо менее явным образом.

Допустим, параметр `UPDATE_FREQUENCY` изначально принимал число секунд.

Со временем, с учетом обратной связи от пользователей, разработчик решил, что длительные интервалы (минуты, часы) неудобно указывать в секундах. Он модифицировал параметр так,

чтобы тот принимал строки (10 минут, 5 секунд и т. п.). Поскольку старые значения параметров не смогут быть обработаны новым контейнером, такое изменение API окажется критическим.

Представьте, что разработчик предусмотрел этот вариант, но сделал так, чтобы значения без единиц измерения интерпретировались как число миллисекунд. Такое изменение, хотя и не приводит к ошибкам, является недопустимым, поскольку приводит к более частым проверкам конфигурации и, как следствие, большей нагрузке на сервер.

Надеюсь, вы осознали, что для обеспечения настоящей модульности необходимо внимательно относиться к предоставляемому вашим контейнером программному интерфейсу. Критические изменения могут быть вызваны менее очевидными, нежели смена имени параметра, причинами.

Документирование контейнеров

На данный момент вы уже умеете параметризовать свои контейнеры, чтобы они были модульными и их можно было повторно использовать. Вы уже знаете, насколько важно поддерживать стабильный программный интерфейс вашего контейнера, чтобы обеспечить его бесперебойную работу у конечных пользователей. Но есть еще один шаг к построению модульных, повторно используемых контейнеров — предоставить пользователям информацию, как их применять в принципе.

Как и в случае с программными библиотеками, ключ к созданию полезной вещи — объяснение, как ею пользоваться. Мало пользы в создании гибкого, надежного, модульного контейнера, если никто не может понять, как с ним работать. К сожалению, на сегодняшний день доступно не так много формальных инструментов, позволяющих документировать образы контейнеров, но есть несколько полезных приемов, упрощающих работу.

У каждого контейнера есть конфигурационный файл `Dockerfile`, на основе которого строится образ контейнера. Именно в нем в первую очередь стоит искать документацию к контейнеру. Некоторые части `Dockerfile` документируют работу контейнера сами по себе. Одним из примеров может служить директива `EXPOSE`, в которой перечислены сетевые порты, открытые в контейнере. Указывать ее не обязательно, но это считается хорошим тоном. Неплохо также снабдить ее комментарием, поясняющим, какой конкретно сервис прослушивает данный порт. Например:

```
...

# Главный сервер принимает запросы на порт 8080
EXPOSE 8080
...
```

Если вы используете переменные среды для параметризации контейнера, то, чтобы установить их значения по умолчанию, можно указать директиву `ENV` с комментарием:

```
...

# Параметр PROXY_PORT обозначает порт, на который необходимо
# перенаправлять запросы
ENV PROXY_PORT 8000
...
```

С помощью директивы `LABEL` к образу можно добавить метаданные — e-mail разработчика, адрес веб-страницы, версию образа и т. д.

```
...

LABEL "org.label-schema.vendor"="name@company.com"
LABEL "org.label.url"="http://images.company.com/my-cool-image"
LABEL "org.label-schema.version"="1.0.3"
...
```

Имена меток метаданных позаимствованы из проекта Label Schema (http://label-schema.org/rc1). Цель проекта — сформировать набор общеизвестных меток.

В случае общей таксономии меток образов разные инструменты могут полагаться на одну и ту же метаинформацию с целью визуализации, мониторинга и корректного использования приложения. При использовании согласованных, общеупотребительных терминов появляется возможность задействовать инструменты, разработанные сообществом, не меняя образа контейнера. Безусловно, можно добавлять и любые другие метки, уместные в контексте вашего образа.

Резюме

В данной главе мы познакомились с паттерном Sidecar, который комбинирует несколько контейнеров на одном вычислительном узле. В рамках паттерна Sidecar контейнер-прицеп дополняет и расширяет контейнер приложения, тем самым добавляя ему функциональности. Sidecar можно использовать для модернизации унаследованных приложений, если внесение изменений в них обойдется слишком дорого. Кроме того, этот паттерн можно применять для создания модульных контейнеров-утилит, задающих стандартную реализацию часто используемых функциональных возможностей. Контейнеры-утилиты можно задействовать во множестве приложений, повышая согласованность среды и снижая расходы на разработку последующих приложений.

Следующие главы познакомят вас с другими паттернами, демонстрирующими иные области применения модульных, повторно используемых контейнеров.

3 Паттерн Ambassador

В предыдущей главе мы рассмотрели паттерн Sidecar, в рамках которого контейнер-прицеп функционально дополняет существующий контейнер приложения. В этой главе мы познакомимся с паттерном Ambassador («посол»). Контейнер-посол выступает посредником во взаимодействии контейнера приложения с внешним миром. Как и в случае с остальными одноузловыми паттернами проектирования, два контейнера составляют симбиотический союз и исполняются совместно на одном компьютере.

Схема данного паттерна изображена на рис. 3.1.

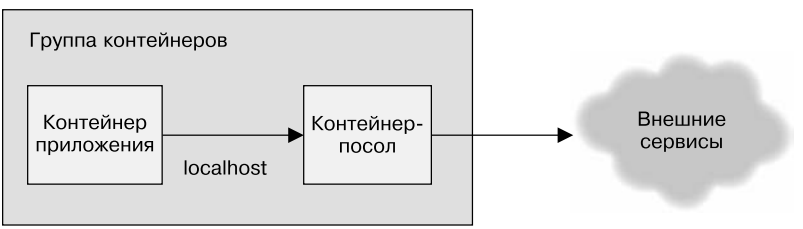

Рис. 3.1. Обобщенный паттерн Ambassador

От паттерна Ambassador двойная польза. Во-первых, как и остальные одноузловые паттерны, он позволяет создавать модульные, повторно используемые контейнеры. Разделение ответственности между контейнерами упрощает их разработку и поддержку. Во-вторых, контейнер-посол можно использовать с различными контейнерами приложений. Такого рода повторное применение ускоряет разработку приложений, поскольку контейнеризованный код можно задействовать в нескольких разных местах. Вдобавок повышаются качество и согласованность реализации, поскольку код собирается разово и затем используется во многих различных контекстах.

В оставшейся части данной главы мы рассмотрим несколько практических примеров реализации паттерна Ambassador.

Использование паттерна Ambassador для шардирования сервиса

В какой-то момент данных на уровне хранилища (storage layer) становится так много, что они перестают помещаться на одной машине. В таких ситуациях необходимо *шардировать* уровень хранилища.

Шардинг (шардирование) — разделение уровня хранилища на несколько независимых частей (шардов), каждая из которых размещается на отдельной машине. В данной главе рассматривается одноузловой паттерн проектирования, предназначенный для адаптирования существующих сервисов, чтобы те могли взаимодействовать с другими, шардированными сервисами, находящимися где-то в Интернете. Здесь не рассматривается, откуда появляются шардированные сервисы. О шардировании и многоузловом паттерне проектирования шардированных сервисов мы подробно поговорим в главе 6.

Схема шардированного сервиса представлена на рис. 3.2.

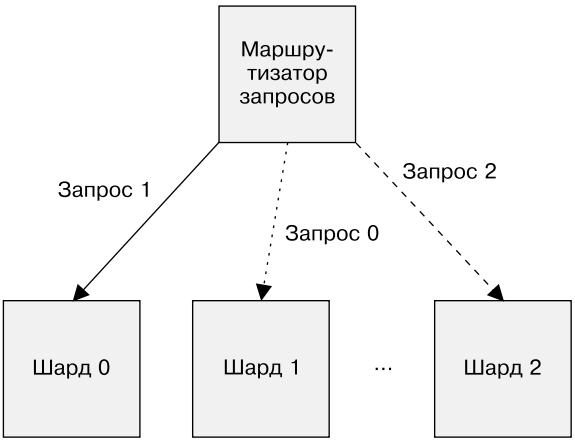

Рис. 3.2. Обобщенная схема шардированного сервиса

При развертывании шардированного сервиса возникает вопрос о том, как интегрировать его с программным обеспечением клиентского или промежуточного уровня. Очевидно, должен существовать модуль, который бы переадресовывал конкретный запрос конкретному шарду. Часто такой шардированный клиент тяжело интегрировать в систему, компоненты которой рассчитывают на подключение к единому хранилищу данных. К тому же шардированные сервисы препятствуют совместному использованию конфигурации средой разработки (где хранилище состоит, как правило, из одного шарда) и средой эксплуатации (где хранилище часто состоит из множества шардов).

Один из вариантов — включить всю логику шардирования в сам шардированный сервис. При таком подходе шардированный сервис должен также иметь балансировщик нагрузки с независимой обработкой транзакций, адресующий трафик нужному шарду. Этот балансировщик нагрузки будет, по сути, распре-

деленной реализацией паттерна Ambassador в виде сервиса. Клиентская реализация паттерна Ambassador становится не нужна, но за счет этого усложняется развертывание шардированного сервиса. Другой вариант — интегрировать одноузловую реализацию паттерна Ambassador на стороне клиента, чтобы она перенаправляла трафик нужному шарду.

Развертывание клиента несколько усложняется, зато упрощается развертывание шардированного сервиса. Как и в случае с любыми компромиссами, особенности вашего конкретного приложения будут определять то, какой из двух подходов окажется наиболее осмысленным. В первую очередь нужно разобраться, каким образом разграничиваются обязанности в вашей команде, во вторую — установить, разрабатываете вы новый код либо развертываете уже существующие решения.

Каждый из подходов верен по-своему. В следующем разделе описывается, как использовать одноузловой паттерн Ambassador для шардирования на стороне клиента.

При адаптировании существующего приложения к шардированному хранилищу мы создаем контейнер-посол, который содержит всю необходимую логику для переадресации запросов соответствующим шардам хранилища. Таким образом, программное обеспечение клиентского или промежуточного уровней подключается к сервису, который выглядит как единое хранилище, работающее на локальной машине. Но этот сервис на самом деле является *шардирующим прокси-контейнером*, реализующим паттерн Ambassador. Он принимает запросы от приложения, переадресует их соответствующему шарду хранилища и затем возвращает результат приложению.

Главный результат применения паттерна Ambassador к шардированным сервисам — разделение обязанностей между контейнером приложения и шардирующим прокси. Контейнер

приложения знает, что ему надо взаимодействовать с сервисом хранения, находящимся на локальном компьютере, а шардирующий прокси содержит только тот код, который отвечает за корректное шардирование запросов.

Как и любую хорошую реализацию одноузлового паттерна, контейнер-посол можно повторно использовать в различных приложениях. Или, как вы увидите в следующем практикуме, в качестве посла может выступать готовый сервис с открытым исходным кодом, что позволит ускорить разработку распределенной системы в целом.

Практикум. Шардируем Redis-хранилище

Redis — высокопроизводительное хранилище типа «ключ — значение», которое можно использовать в качестве кэша или более постоянного места хранения данных. В этом примере мы задействуем его в качестве кэша. Начнем с развертывания шардированного Redis на кластер Kubernetes. Для этого обратимся к API `StatefulSet`, поскольку он дает каждому шарду уникальные DNS-имена, которыми мы воспользуемся при настройке прокси.

`StatefulSet` для Redis будет выглядеть следующим образом:

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: sharded-redis
spec:
  serviceName: "redis"
  replicas: 3
  template:
    metadata:
      labels:
        app: redis
    spec:
```

```
      terminationGracePeriodSeconds: 10
      containers:
      - name: redis
        image: redis
        ports:
        - containerPort: 6379
          name: redis
```

Сохраните этот код в файле `redis-shards.yaml`, который можно развернуть командой `kubectl create -f redis-shards.yaml`. Она создаст три контейнера с запущенным Redis. Их можно увидеть, выполнив команду `kubectl get pods`. Результат будет таким:

```
sharded-redis-[0,1,2]
```

Очевидно, недостаточно лишь запустить несколько копий Redis — нам также необходимы имена, по которым к ним можно обращаться. Воспользуемся для этого Kubernetes Service, который назначит DNS-имена созданным репликам. Он будет выглядеть следующим образом:

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

Сохраните этот код в файле `redis-shards.yaml` и разверните полученный файл командой `kubectl create -f redis-shards.yaml`. Теперь у вас должны появиться DNS-записи для `sharded-redis-0.redis`, `sharded-redis-1.redis` и т. д. Воспользуемся этими именами для настройки прокси-сервера twemproxy.

Это легковесный, высокопроизводительный прокси-сервер для memcached и Redis, который изначально был разработан в Twitter. Его исходный код теперь доступен на GitHub (https://github.com/twitter/twemproxy).

Следующая конфигурация настроит twemproxy на использование созданных нами копий Redis.

```
redis:
  listen: 127.0.0.1:6379
  hash: fnv1a_64
  distribution: ketama
  auto_eject_hosts: true
  redis: true
  timeout: 400
  server_retry_timeout: 2000
  server_failure_limit: 1
  servers:
    - sharded-redis-0.redis:6379:1
    - sharded-redis-1.redis:6379:1
    - sharded-redis-2.redis:6379:1
```

Из конфигурационного файла видно, что запросы к Redis обслуживаются по адресу localhost:6379, так что контейнер приложения может получить доступ к контейнеру-послу. Его можно развернуть в «посольскую» группу контейнеров, используя Kubernetes-объект `ConfigMap`, который можно создать такой командой:

```
kubectl create configmap twem-config --from-file=./nutcracker.yaml
```

Подготовительные мероприятия завершены, и можно развернуть наш пример реализации паттерна Ambassador. Определяем группу контейнеров следующим образом:

```
apiVersion: v1
kind: Pod
metadata:
  name: ambassador-example
```

```
spec:
  containers:
    # Сюда необходимо подставить имя контейнера приложения, например:
    # - name: nginx
    # image: nginx
    # Здесь указываем имя контейнера-посла
    - name: twemproxy
      image: ganomede/twemproxy
      command:
      - "nutcracker"
      - "-c"
      - "/etc/config/nutcracker.yaml"
      - "-v"
      - "7"
      - "-s"
      - "6222"
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
    configMap:
      name: twem-config
```

Сначала в группе объявляется контейнер-посол, а конкретный контейнер приложения может быть внедрен в нее позже.

Использование паттерна Ambassador для реализации сервиса-посредника

В попытке обеспечить переносимость приложения между разными средами (публичным облаком, физическим центром обработки данных либо частным облаком) основной проблемой становится обнаружение и конфигурирование сервисов. Чтобы понять, что это значит, представьте себе клиентский модуль, хранящий данные в базе данных MySQL. В публичном облаке такая база предоставлялась бы по схеме «ПО как сервис» (software-as-a-service, SaaS). В частном облаке, однако же, может

потребоваться динамически «поднять» новую виртуальную машину или контейнер с MySQL.

Следовательно, переносимое приложение должно иметь возможность изучить свое окружение и найти подходящий MySQL-сервер. Такой процесс называется *обнаружением сервисов* (service discovery), а система, которая выполняет обнаружение и стыковку, называется *сервисом-посредником* (service broker). Как и в предыдущих примерах, использование паттерна Ambassador позволяет отделить логику контейнера приложения от логики контейнера-посла сервиса-посредника. Приложение просто подключается к экземпляру сервиса (например, MySQL), работающему на локальном компьютере. Обязанность контейнера-посла сервиса-посредника заключается в обследовании окружения и опосредовании подключения к конкретному экземпляру целевого сервиса. Данный процесс показан на рис. 3.3.

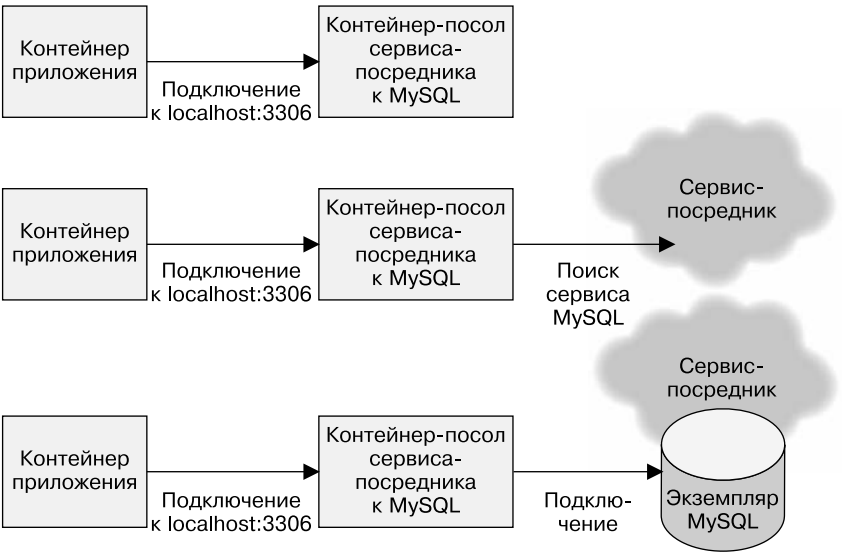

Рис. 3.3. Контейнер-посол сервиса-посредника создает экземпляр MySQL-сервиса

Использование паттерна Ambassador для проведения экспериментов и разделения запросов

Последний пример приложения с использованием паттерна Ambassador — реализация экспериментов и других видов разделения запросов. В эксплуатируемых системах важно иметь возможность разделения запросов, когда некоторая часть запросов обрабатывается не основным рабочим сервисом, а его альтернативной реализацией. Чаще всего его используют, чтобы проводить эксперименты с новыми или бета-версиями сервиса для определения степени надежности и производительности новой версии сервиса по сравнению с существующей.

Кроме того, разделение запросов иногда используется для дублирования или деления трафика таким образом, что он распределяется как на рабочую версию ПО, так и на новую, еще не развернутую.

Ответы рабочей системы возвращаются пользователю, а ответы новой версии сервиса игнорируются. Чаще всего такое деление запросов применяется, когда необходимо сымитировать реальную рабочую нагрузку на сервис, не рискуя повлиять на работу пользователей текущей версии.

С учетом предыдущих примеров становится очевидным то, как делитель запросов может взаимодействовать с контейнером приложения для выполнения своей функции.

Как и ранее, контейнер приложения просто подключается к сервису, работающему на локальном компьютере. Контейнер-посол, в свою очередь, получает запросы, проксирует их как рабочей, так и экспериментальной системам, а затем возвращает ответы рабочей системы так, как будто сам сделал всю работу.

Такое разделение ответственности позволяет четко ограничить функциональность и размер кода приложений, содержащихся в контейнерах. Разделение приложения на модули дает возможность использовать контейнер с делителем запросов во многих разных приложениях и с различными настройками.

Практикум. Реализация 10%-ных экспериментов

Для эксперимента с разделением запросов воспользуемся веб-сервером nginx. Nginx — мощный, многофункциональный веб-сервер с открытым исходным кодом. Чтобы сконфигурировать nginx для применения в контейнере-после, воспользуемся следующей конфигурацией (обратите внимание, что она рассчитана на HTTP, но ее легко адаптировать под HTTPS):

```
worker_processes 5;
error_log error.log;
pid nginx.pid;
worker_rlimit_nofile 8192;

events {
  worker_connections 1024;
}

http {
  upstream backend {
    ip_hash;
    server web weight=9;
    server experiment;
  }

  server {
    listen localhost:80;
    location / {
      proxy_pass http://backend;
    }
  }
}
```

Как и в случае с упомянутыми ранее шардированными сервисами, можно развернуть инструментарий для экспериментов в виде отдельного микросервиса, а не интегрировать его как часть клиентской pod-группы. Делая так, вы добавляете еще один сервис, который нужно поддерживать, масштабировать, мониторить и т. д. Если есть вероятность, что экспериментирование укоренится в вашей архитектуре, в таком решении может быть смысл. Если оно применяется скорее время от времени, то более осмысленным будет использование контейнера-посла на стороне клиента.

Обратите внимание, что в данной конфигурации я использую хеширование IP. Оно необходимо для того, чтобы пользователь не задействовал попеременно то основную, то тестовую версию сервиса. Благодаря этому пользователи будут взаимодействовать с приложением единообразно.

Весовой коэффициент используется, чтобы 90 % трафика приходилось на основное приложение, а 10 % — на тестовую версию.

Как и в других примерах, мы будем разворачивать данную конфигурацию как объект `ConfigMap` в Kubernetes.

```
kubectl create configmaps --from-file=nginx.conf
```

Она исходит из того, что сервисы web и experiment уже определены. Если это не так, то вам необходимо их создать до создания контейнера-посла, поскольку nginx не запустится корректно, если не сможет найти сервисы, которым он проксирует запросы.

Вот несколько примеров конфигурации:

```
# Сервис 'experiment'
apiVersion: v1
kind: Service
metadata:
  name: experiment
```

```
  labels:
    app: experiment
spec:
  ports:
  - port: 80
    name: web
  selector:
    # Установите значение данного селектора в соответствии
    # с метками вашего приложения
    app: experiment
---
# Сервис 'prod'
apiVersion: v1
kind: Service
metadata:
  name: web
  labels:
    app: web
spec:
  ports:
  - port: 80
    name: web
  selector:
    # Установите значение этого селектора в соответствии
    # с метками вашего приложения
    app: web
```

Затем разворачиваем nginx в роли посла в рамках контейнера:

```
apiVersion: v1
kind: Pod
metadata:
  name: experiment-example
spec:
  containers:
    # Сюда необходимо подставить имя контейнера приложения,
    # например:
    # - name: some-name
    # image: some-image
    # Здесь указываем имя контейнера-посла
    - name: nginx
      image: nginx
```

```
      volumeMounts:
      - name: config-volume
        mountPath: /etc/nginx
  volumes:
    - name: config-volume
      configMap:
        name: experiment-config
```

Чтобы воспользоваться контейнером-послом в полной мере, в группу можно добавить еще один или несколько контейнеров.

4 Адаптеры

В предыдущих главах мы рассмотрели, как с помощью паттерна Sidecar расширять и дополнять существующие контейнеры приложений. Мы также разобрали, как контейнеры-послы могут опосредовать и даже изменять способ взаимодействия контейнера с внешним миром. В этой главе описывается последний одноузловой паттерн — *Adapter*. В его рамках *контейнер-адаптер* модифицирует программный интерфейс *контейнера приложения* таким образом, чтобы он соответствовал некоему заранее определенному интерфейсу, реализация которого ожидается от всех контейнеров приложений. К примеру, адаптер может обеспечивать реализацию унифицированного интерфейса мониторинга. Или же он может обеспечивать то, что файлы журнала всегда выводятся в stdout, а также требовать соблюдения любых других соглашений.

Разработка реальных приложений — тренировка по построению гетерогенных, гибридных систем. Одни части вашего приложения могут быть написаны вашей командой с нуля, другие

получены от поставщиков, третьи — вообще быть готовыми проприетарными решениями или решениями с открытым кодом, используемыми в виде двоичных исполняемых файлов. Совокупный эффект такой гетерогенности состоит в том, что любое реальное приложение, которое вам приходилось или придется развертывать, написано на множестве языков и с учетом разных соглашений относительно ведения файлов журнала, мониторинга и других подобных общих задач.

Но для того, чтобы эффективно следить за работой приложения и управлять им, нужны общие интерфейсы. Когда каждое приложение выводит показатели в разных форматах посредством разных интерфейсов, очень тяжело собирать их для визуализации и уведомления о нештатных ситуациях. Именно в такой ситуации и уместен паттерн Adapter. Как и другие одноузловые паттерны, паттерн Adapter состоит из модульных контейнеров. Разные контейнеры приложений могут предоставлять разные интерфейсы для мониторинга, а контейнер-адаптер подстраивается под гетерогенность среды с целью унификации интерфейса. Это позволяет разворачивать единственный инструмент, заточенный под этот конкретный интерфейс. Рисунок 4.1 обобщенно иллюстрирует данный паттерн.

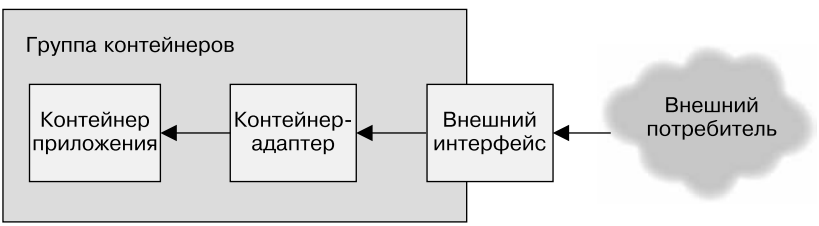

Рис. 4.1. Обобщенный паттерн Adapter

Далее в главе мы рассмотрим несколько приложений паттерна Adapter.

Мониторинг

Хотелось бы иметь унифицированное решение, позволяющее автоматически обнаруживать любые приложения, развернутые в некоторой среде, и наблюдать за их состоянием. Чтобы это стало возможным, каждое приложение должно реализовывать один и тот же интерфейс мониторинга.

Существует множество стандартизированных интерфейсов мониторинга — `syslog`, мониторинг событий Windows (ETW), JMX для Java-приложений и многие другие протоколы и интерфейсы. Однако все они различаются как протоколами, так и способами коммуникации (push или pull).

К сожалению, приложения, входящие в вашу распределенную систему, могут включать в себя как части из самописного кода, так и готовые open-source-компоненты. В результате вы сталкиваетесь с широким разнообразием интерфейсов мониторинга, которые необходимо интегрировать в одну понятную систему.

К счастью, разработчики большинства решений, касающихся мониторинга, осознают, что эти решения должны быть широко применимы, и поэтому реализуют механизм надстроек, позволяющий адаптировать формат мониторинга к некоторому общему интерфейсу. Как развертывать приложения гибким и устойчивым образом, имея такой набор инструментов? У паттерна Adapter есть ответ на данный вопрос. Применительно к мониторингу мы видим, что контейнер приложения — это просто приложение, за которым мы хотим наблюдать.

Контейнер-адаптер содержит инструменты, преобразующие интерфейс мониторинга, предоставляемого контейнером приложения, в интерфейс, ожидаемый системой мониторинга общего назначения.

Разбиение системы таким образом позволяет создавать более понятные и легко сопровождаемые системы. Развертывание новых версий приложения не требует развертывания новой версии адаптера для мониторинга. К тому же контейнер-монитор может быть повторно использован совместно с разными контейнерами приложений. Кроме того, его может даже предоставить независимая команда, занимающаяся поддержкой подсистемы мониторинга. Наконец, развертывание адаптера для мониторинга в виде отдельного контейнера гарантирует, что каждый контейнер получит свой выделенный набор ресурсов: процессор, память и т. п. Благодаря этому неправильно функционирующий контейнер не вызовет проблем с пользовательскими сервисами.

Практикум. Мониторинг с помощью Prometheus

В качестве примера рассмотрим мониторинг состояния контейнеров с помощью Prometheus (https://prometheus.io/). Prometheus — это агрегатор данных мониторинга, который собирает показатели и организует в упорядоченную по времени базу данных. Кроме базы данных, Prometheus предоставляет средства визуализации и язык запросов для анализа собранных показателей. Чтобы обеспечить сбор показателей из разных систем, Prometheus рассчитывает, что у каждого контейнера предусмотрен определенный программный интерфейс для сбора показателей. Это позволяет Prometheus следить за самыми разными приложениями через единообразный интерфейс.

Однако многие приложения, такие как хранилище «ключ — значение» Redis, предоставляют показатели в формате, не совместимом с Prometheus. Следовательно, паттерн Adapter весьма полезен для адаптирования существующих сервисов вроде Redis к интерфейсу сбора показателей Prometheus.

Рассмотрим спецификацию простой группы контейнеров для сервиса Redis:

```
apiVersion: v1
kind: Pod
metadata:
  name: adapter-example
  namespace: default
spec:
  containers:
    - image: redis
      name: redis
```

На данный момент Prometheus не может наблюдать за состоянием такого контейнера, поскольку тот не предоставляет нужного интерфейса. Однако если мы просто добавим контейнер-адаптер (в данном случае открытый инструмент экспорта в формат Prometheus), то сможем модифицировать эту группу так, чтобы она предоставляла необходимый интерфейс, соответствующий ожиданиям Prometheus.

```
apiVersion: v1
kind: Pod
metadata:
  name: adapter-example
  namespace: default
spec:
  containers:
  - image: redis
    name: redis
  # Предоставляем адаптер, реализующий интерфейс Prometheus
  - image: oliver006/redis_exporter
    name: adapter
```

Этот пример иллюстрирует не только ценность паттерна Adapter в плане обеспечения унифицированного интерфейса, но и полезность контейнерных паттернов в целом как средства обеспечения модульности и повторного использования контейнеров. Пример демонстрирует, как совместить существующий контейнер с Redis и адаптер в формат Prometheus. Совокупным эффектом

от их стыковки станет Redis-сервер с возможностью мониторинга при минимуме усилий с нашей стороны. В отсутствие паттерна Adapter развертывание такой же функциональности потребовало бы гораздо больше наших собственных усилий, привело бы к менее управляемому результату, поскольку любое обновление Redis или адаптера требует дополнительных трудозатрат на обновление.

Ведение журналов

Как и в случае с мониторингом, системы очень неоднородно журналируют данные. Системы могут разделять журналы на различные уровни, например debug, info, warning и error, каждый из которых записывается в отдельный файл. Некоторые просто выводят информацию в потоки `stdout` или `stderr`. Это особенно критично в случае контейнеризованных приложений, когда обычно ожидается, что контейнеры выводят информацию в поток `stdout`, так как именно его содержимое доступно при выполнении команд `docker logs` или `kubectl logs`.

Усложняет ситуацию и то, что журналируемая информация в общем случае имеет структурированные элементы, например дату и время записи, но эти сведения сильно различаются для разных реализаций библиотек журналирования (например, для встроенного в Java средства журналирования и пакета `glog` в Go).

Записывая и читая журналы своей распределенной системы, вы, конечно же, не особо заботитесь о различиях между форматами. Вы хотите убедиться, что, несмотря на различную структуру данных, каждая запись имеет соответствующую метку времени.

К счастью, как и в случае мониторинга, паттерн Adapter помогает предоставить модульную, повторно используемую архитектуру для ведения журналов. Контейнер приложения может вести журнал в файле, а контейнер-адаптер будет перенаправлять его содержимое в поток `stdout`. Разные контейнеры приложения

могут вести журналы в разных форматах, а контейнер-адаптер может преобразовывать эти данные в общее структурированное представление, которым сможет воспользоваться агрегатор журналов. Адаптер и в данном случае на основе неоднородной среды приложений создает однородную среду общих интерфейсов.

При планировании использования контейнеров-адаптеров часто возникает один вопрос: почему бы не изменять сам контейнер приложения? Если вы разработчик, отвечающий за контейнер приложения, то это может оказаться хорошим решением. Адаптация самого приложения или его контейнера с целью обеспечения унифицированного интерфейса — хорошая идея.

Однако во многих случаях приходится пользоваться контейнером, созданным другим человеком. В таких случаях создание преобразованного образа, который придется поддерживать (выпускать собственные исправления, следить за исправлениями, выпускаемыми автором исходного образа), оказывается более затратным, чем разработка собственного адаптера, который работает параллельно «чужому» контейнеру. Кроме того, выделение адаптера в отдельный контейнер позволяет использовать его повторно в других приложениях, что невозможно, если модифицировать непосредственно контейнер приложения.

Практикум. Нормализация форматов журналов с помощью fluentd

Одна из задач адаптера — привести показатели журнала к стандартному набору событий. Разные приложения имеют разные форматы выходных данных, но, чтобы привести их к однородному формату, можно задействовать стандартный инструмент журналирования, развернутый в виде адаптера. В данном примере мы будем использовать агент мониторинга fluentd, а также некоторые поддерживаемые сообществом надстройки для получения записей журналов из различных источников.

Fluentd (https://fluentd.org/) — один из наиболее популярных агентов журналирования с открытым исходным кодом. Одно из его основных преимуществ — богатый набор поддерживаемых сообществом надстроек, которые обеспечивают гибкий мониторинг разнообразных приложений.

Для начала понаблюдаем за сервисом Redis. Redis — популярное хранилище типа «ключ — значение». В числе прочих он предоставляет команду `SLOWLOG`, которая перечисляет последние запросы, превысившие определенный порог времени исполнения. Такая информация чрезвычайно полезна при отладке проблем с производительностью вашего приложения. К сожалению, инструмент `SLOWLOG` доступен только в виде команды сервера Redis, а это означает, что такие проблемы сложно отлаживать ретроспективно в том случае, когда нет возможности сделать это сразу. Чтобы преодолеть это ограничение, можно воспользоваться сервисом fluentd и паттерном Adapter, добавляя тем самым в Redis возможность журналирования медленных запросов.

Для этого воспользуемся паттерном Adapter, в рамках которого назначим контейнер сервиса Redis основным контейнером приложения, а контейнер сервиса fluentd — контейнером-адаптером. Чтобы следить за медленными запросами, мы также воспользуемся надстройкой fluent-plugin-redis-slowlog (https://github.com/mominosin/fluent-plugin-redis-slowlog). Сконфигурировать ее можно, как показано в следующем фрагменте:

```
<source>
  type redis_slowlog
  host localhost
  port 6379
  tag redis.slowlog
</source>
```

Мы используем адаптер, а контейнеры находятся в общем сетевом пространстве, — конфигурация журналирования ограничивается настройкой на сервер localhost и порт Redis по умолчанию (6379).

Благодаря такому приложению паттерна Adapter журнал медленно выполняющихся запросов будет доступен в любой удобный для их отладки момент.

В качестве похожего упражнения можно настроить наблюдение за записями журнала системы Apache Storm (https://storm.apache.org/). Storm предоставляет данные посредством RESTful API, что само по себе удобно, но имеет ограничения в том случае, когда мы не наблюдаем за системой в момент возникновения проблемы. Как и в случае с Redis, можно воспользоваться fluentd-адаптером, чтобы преобразовать данные Storm в упорядоченный по времени журнал, к которому можно осуществлять запросы. Чтобы добиться этого, можно развернуть fluentd-адаптер с развернутой в нем надстройкой fluent-plugin-storm.

Надстройку стоит сконфигурировать таким образом, чтобы она указывала на сервер localhost, поскольку опять-таки мы работаем с группой контейнеров с общим сетевым пространством. Конфигурационный файл надстройки будет выглядеть так:

```
<source>
  type storm
  tag storm
  url http://localhost:8080
  window 600
  sys 0
</source>
```

Мониторинг работоспособности сервисов

В качестве последнего примера рассмотрим применение паттерна Adapter для мониторинга работоспособности контейнера приложения. Разберем задачу мониторинга работоспособности контейнера типовой СУБД. В данном случае контейнер СУБД предоставляется ее разработчиками, и мне не хотелось бы модифицировать его

только для того, чтобы добавить в него проверку работоспособности. Оркестратор контейнеров, конечно, может взять на себя несложную проверку работоспособности, чтобы убедиться, что процесс запущен и принимает подключения по определенному порту. А что, если мы хотим выполнять сложную проверку работоспособности, делая, например, запросы к базе данных?

Оркестраторы контейнеров наподобие Kubernetes также позволяют задавать сценарии оболочки, проверяющие работоспособность контейнера. Имея такую возможность, мы можем написать сложный сценарий оболочки, выполняющий несколько диагностических запросов к базе данных, чтобы определить степень ее работоспособности. Но где хранить такой сценарий и как следить за его версиями?

Нетрудно догадаться, как решить эту проблему, — можно использовать контейнер-адаптер. База данных работает в контейнере приложения, который имеет общий с контейнером-адаптером сетевой интерфейс. Контейнер-адаптер — простой контейнер, который содержит только сценарий оболочки, оценивающий работоспособность базы данных. Этот сценарий можно настроить в качестве комплексного средства проверки контейнера СУБД, выполняющего любую диагностику, необходимую нашему приложению. Если контейнер приложения когда-либо не пройдет проверку, он будет автоматически перезапущен.

Практикум. Комплексный мониторинг работоспособности MySQL

Допустим, вы хотите следить за работоспособностью базы данных MySQL путем периодического выполнения запросов, соответствующих рабочей нагрузке вашего приложения. Одним из вариантов будет добавить в контейнер MySQL проверку работоспособности, отвечающую вашим требованиям. В общем случае, однако, это не очень желательное решение, поскольку

оно требует преобразования базового MySQL-образа, а также обновления модифицированного образа по мере выхода новых версий базового образа.

В этом случае использование паттерна Adapter гораздо более привлекательно, нежели добавление проверок работоспособности непосредственно в базовый образ. Вместо того чтобы модифицировать существующий контейнер с MySQL, можно добавить к типовому MySQL-контейнеру контейнер-адаптер, который бы проверял состояние базы данных. Учитывая, что адаптер проверяет работоспособность посредством протокола HTTP, все сводится к определению процесса проверки работоспособности базы данных в терминах интерфейса, предоставляемого адаптером.

Исходный текст такого адаптера довольно прост и выглядит на языке Go следующим образом (очевидно, что его можно реализовать и на другом языке):

```
package main

import (
    "database/sql"
    "flag"
    "fmt"
    "net/http"

    _ "github.com/go-sql-driver/mysql"
)

var (
    user   = flag.String("user", "", "Имя пользователя базы данных")
    passwd = flag.String("password", "", "Пароль к базе данных")
    db     = flag.String("database", "", "К какой базе данных
                          необходимо подключиться")
    query  = flag.String("query", "", "Тестовый запрос")
    addr   = flag.String("address", "localhost:8080",
                          "По какому IP-адресу принимать запросы")
)
```

```
// Пример использования:
//   db-check --query="SELECT * from my-cool-table" \
//            --user=bdburns \
//            --passwd="you wish"
//
func main() {
    flag.Parse()
    db, err := sql.Open("localhost", fmt.Sprintf("%s:%s@/%s",
                        *user, *passwd, *db))
    if err != nil {
        fmt.Printf("Ошибка подключения к базе данных: %v", err)
    }

    // Простой веб-обработчик, выполняющий запрос
    http.HandleFunc("", func(res http.ResponseWriter,
                    req *http.Request) {
        _, err := db.Exec(*query)
        if err != nil {
            res.WriteHeader(http.StatusInternalServerError)
            res.Write([]byte(err.Error()))
            return
        }
        res.WriteHeader(http.StatusOK)
        res.Write([]byte("OK"))
        return
    })
    // Запуск сервера
    http.ListenAndServe(*addr, nil)
}
```

Затем мы можем собрать контейнер-адаптер и поместить его в группу, которая будет выглядеть следующим образом:

```
apiVersion: v1
kind: Pod
metadata:
  name: adapter-example-health
  namespace: default
spec:
  containers:
  - image: mysql
    name: mysql
```

```
- image: brendanburns/mysql-adapter
  name: adapter
```

Контейнер `mysql` остается неизменным, при этом необходимую обратную связь можно получить от контейнера-адаптера.

На первый взгляд может показаться, что такой вариант применения паттерна Adapter является излишним. Мы, конечно, можем собрать свой собственный образ, который знает, как проверять работоспособность экземпляра `mysql`. Это верно, но подобный подход игнорирует преимущества, следующие из модульности. Если каждый разработчик будет реализовывать свою собственную модификацию контейнера со встроенной проверкой работоспособности, то потеряется возможность повторного (или совместного) его использования.

Напротив, если применять паттерны вроде Adapter для разработки модульных решений, состоящих из нескольких контейнеров, то приложение естественным образом декомпозируется на части, которые можно использовать повторно. Адаптер, разработанный для проверки работоспособности `mysql`, может быть совместно/повторно использован многими людьми. Кроме того, разработчики могут применять паттерн Adapter, используя общий контейнер для проверки работоспособности, не вдаваясь в детали наблюдения за базами данных `mysql`. Таким образом, модульность в целом и паттерн Adapter в частности не только способствуют совместному применению кода, но и позволяют воспользоваться знаниями других людей.

Надо отметить, что паттерны проектирования предназначены не только для их непосредственного применения в приложениях, но и для развития сообществ, участники которых могут взаимодействовать между собой и делиться результатами.

Часть II
Паттерны проектирования обслуживающих систем

В предыдущей главе мы рассмотрели паттерны группирования наборов контейнеров, совместно исполняемых на одной машине. Подобные группы представляют собой тесно связанные, симбиотические системы. Они зависят от совместно используемых локальных ресурсов: дискового пространства, сетевых интерфейсов, а также от межпроцессного взаимодействия. Такие наборы контейнеров являются не только важными паттернами, но и строительными блоками для более крупных систем. Требования к надежности, масштабируемости, разделению обязанностей обуславливают то, что реальные системы состоят из множества различных компонентов, развернутых на многих машинах. Компоненты в многоузловых паттернах связаны слабее, чем в одноузловых. Хотя эти паттерны и диктуют схему взаимодействия компонентов между собой, само взаимодействие осуществляется через сетевые вызовы. Кроме того, параллельно выполняется множество вызовов, координируемых путем нестрогой синхронизации, а не с помощью ограничений реального времени.

Введение в микросервисы

С недавних пор термин *«микросервисы»* стал модным словечком, описывающим системы с многоузловыми распределенными архитектурами. Микросервисами называются системы, созданные из множества разных компонентов, работающих в разных процессах и взаимодействующих посредством заранее определенных программных интерфейсов. Микросервисы противопоставляются *монолитным* системам, которые

сосредотачивают функциональность сервиса в одном строго скоординированном приложении. Эти два подхода изображены на рис. II.1 и II.2.

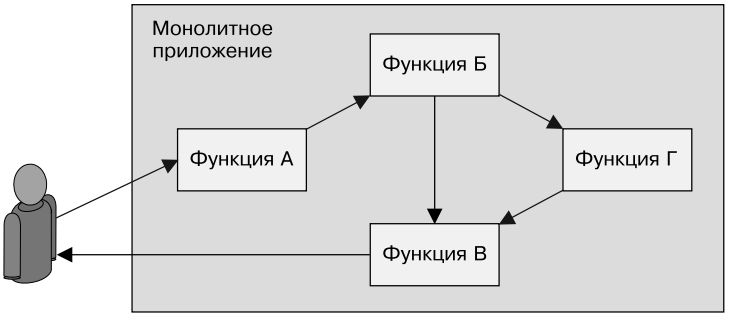

Рис. II.1. Монолитный сервис, вся функциональность которого сосредоточена в одном контейнере

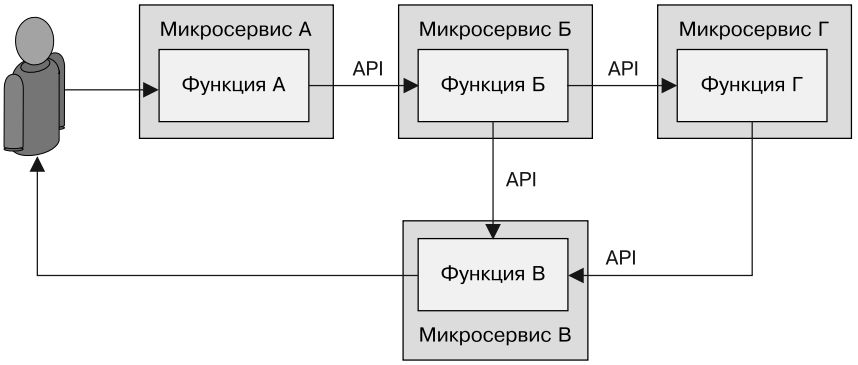

Рис. II.2. Микросервисная архитектура, в которой под каждую функцию выделяется отдельный контейнер

Микросервисный подход имеет немало преимуществ, многие из которых связаны с надежностью и гибкостью. Микросервисы делят приложение на небольшие части, каждая из которых отвечает за предоставление определенной услуги. За счет сужения

области действия сервисов каждый сервис в состоянии разрабатывать и поддерживать команда, которую можно накормить двумя пиццами[1]. Уменьшение размера команды также снижает расходы на поддержание ее деятельности.

Кроме того, появление формального интерфейса между микросервисами ослабляет взаимозависимость команд и устанавливает надежный контракт между сервисами. Такой формальный контракт снижает потребность в тесной синхронизации команд, поскольку команда, предоставляющая API, понимает, в каком объеме необходимо обеспечивать совместимость, а команда, потребляющая API, может рассчитывать на стабильное обслуживание, не заботясь о деталях реализации потребляемого сервиса. Такая декомпозиция позволяет командам независимо управлять темпом разработки и графиком выпуска новых версий, что дает им возможность выполнять итерации, улучшая тем самым код сервиса.

Наконец, разделение на микросервисы повышает масштабируемость. Поскольку каждый компонент выделен в отдельный сервис, его можно масштабировать независимо от остальных. Нечасто случается так, что все сервисы в рамках более крупного приложения развиваются в одном темпе и масштабируются одинаковым образом. Некоторые системы не имеют внутреннего состояния, и их можно масштабировать горизонтально, в то же время в других системах оно есть и требует шардирования или других подходов к масштабированию. Когда сервисы отделены друг от друга, каждый из них можно масштабировать наиболее подходящим способом. Это невозможно, когда все сервисы являются частями большого монолитного приложения.

Микросервисный подход к проектированию систем, безусловно, имеет и свои недостатки. Два наиболее очевидных недостатка

---

[1] Two-pizza team — термин, введенный главой Amazon Дж. Безосом (J. Bezos) для небольших команд с тесной внутренней коммуникацией.

состоят в том, что связи внутри системы становятся слабее, а значит, отладка системы в случае отказа становится намного сложнее. Больше не получится загрузить в отладчик одно приложение и выяснить, что идет не так. Любая ошибка оказывается следствием того, что большое количество систем работает на большом количестве машин. Такую среду сложно воспроизвести в отладчике. Неизбежным итогом этого является также и то, что микросервисные системы сложно проектировать и отлаживать. Системы, основанные на микросервисах, используют различные способы и схемы взаимодействия между сервисами (синхронный, асинхронный, передача сообщений и т. п.), а также множество различных паттернов координации и управления сервисами.

Эти проблемы обуславливают потребность в распределенных паттернах. Когда микросервисная архитектура состоит из хорошо известных паттернов, ее проще проектировать, поскольку многие принципы проектирования уже закодированы в паттернах. Кроме того, паттерны упрощают отладку систем, поскольку позволяют разработчикам применять опыт, полученный при отладке других систем, спроектированных с использованием таких же паттернов.

В этой части вы познакомитесь с несколькими многоузловыми паттернами построения распределенных систем. Они не являются взаимно исключающими. Любая реальная система строится на основе набора паттернов, взаимодействующих в рамках одного высокоуровневого приложения.

5 Реплицированные сервисы с распределением нагрузки

Реплицированный сервис с распределением нагрузки — простейший распределенный паттерн, известный многим. В рамках такого сервиса все серверы идентичны друг другу и поддерживают входящий трафик. Паттерн состоит из масштабируемого набора серверов, находящихся за балансировщиком нагрузки. Балансировщик обычно распределяет нагрузку либо по карусельному (round-robin) принципу, либо с применением некоторой разновидности закрепления сессий. В данной главе приводится конкретный пример развертывания такого сервиса с помощью Kubernetes.

Сервисы без внутреннего состояния

Сервисы без внутреннего состояния (stateless-сервисы) не требуют для своей работы сохранения состояния. В простейших приложе-

ниях, не хранящих состояние (stateless-приложениях), отдельные запросы могут быть направлены разным экземплярам сервиса. Примеры stateless-сервисов включают как сервисы доставки статического контента, так и сложные промежуточные системы, принимающие и агрегирующие запросы от серверных сервисов.

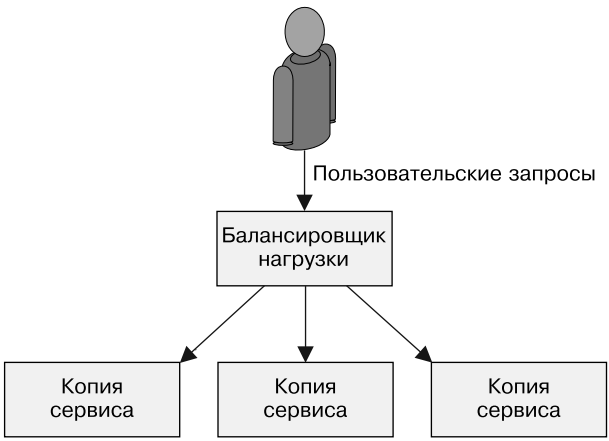

Рис. 5.1. Простой реплицированный stateless-сервис

Системы, не хранящие состояние, реплицируются для обеспечения избыточности и масштабируемости. Сколь угодно малый сервис требует минимум двух копий для обеспечения уровня «высокой доступности» соглашения об уровне услуг (SLA). Чтобы понять, почему это так, рассмотрим предоставление сервиса с *уровнем доступности «три девятки»* (99,9 %). У сервиса с уровнем доступности 99,9 % в день есть 1,4 минуты на простой ($24 \times 60 \times 0{,}001$). Даже исходя из того, что ваш сервис никогда не отказывает, у вас есть не более 1,4 минуты на обновление программного обеспечения, чтобы соблюсти уровень обслуживания при использовании одного экземпляра сервиса. И это с учетом, что вы обновляете программное обеспечение раз в день. Если ваша команда серьезно вступила на путь Continuous Delivery и вы выпускаете новую версию приложения каждый час, то

у вас есть не более *3,6 секунды* на развертывание очередной версии, чтобы соблюсти уровень обслуживания 99,9 % при использовании одного экземпляра сервиса. Стоит чуть задержаться — и вы уже не вписываетесь в заявленные 0,1 % времени простоя.

Можно, конечно, вместо этого всего добавить вторую копию сервиса и балансировщик нагрузки. Таким образом, пока вы развертываете новую версию или восстанавливаете один экземпляр сервиса после сбоя (что, надеюсь, происходит нечасто), ваших ничего не подозревающих пользователей будет обслуживать второй его экземпляр.

По мере роста сервиса ему требуются дополнительные экземпляры для поддержки большего количества одновременно подключенных пользователей. *Горизонтально масштабируемые* системы поддерживают растущее количество пользователей путем добавления дополнительных копий сервиса (рис. 5.2). Это происходит благодаря использованию паттерна Load-balanced Replicated Serving (обслуживание с репликацией и балансировкой нагрузки).

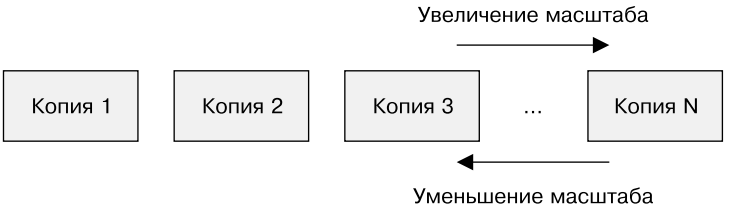

Рис. 5.2. Горизонтальное масштабирование реплицированного stateless-приложения

Датчики готовности для балансировщика нагрузки

Репликация и балансировка нагрузки, конечно же, лишь часть паттерна реплицированного stateless-сервиса. При проектировании

реплицируемого сервиса настолько же важно разработать и развернуть датчик готовности, к которому бы обращался балансировщик нагрузки. Мы уже рассматривали, как можно использовать датчики работоспособности в оркестраторах контейнеров, чтобы своевременно перезапускать приложения. *Датчик готовности* же определяет, готово ли приложение обслужить запрос пользователя. Необходимость такого различия обусловлена тем, что многие приложения требуют некоторого времени на инициализацию, прежде чем они смогут обслуживать пользовательские запросы. Им, возможно, надо подключиться к базе данных, загрузить надстройки либо загрузить файлы из сети. Во всех этих случаях контейнеры *исправны*, но *не готовы*. При построении приложения, использующего паттерн Replicated Service, не забывайте предусмотреть специальный URL, реализующий проверку готовности.

Практикум. Создание реплицированного сервиса с помощью Kubernetes

В следующем примере приводится инструкция по развертыванию реплицированного stateless-сервиса с балансировщиком нагрузки. Эта инструкция подразумевает использование оркестратора контейнеров Kubernetes, но сам паттерн можно реализовать и с помощью других оркестраторов.

Для начала создадим простое NodeJS-приложение, которое выводит толкования слов из словаря.

Сервис можно опробовать в контейнере, выполнив следующую команду:

```
docker run -p 8080:8080 brendanburns/dictionary-server
```

Она запустит простой словарный сервер на локальном компьютере. Например, чтобы узнать толкование слова dog, необходимо перейти по адресу http://localhost:8080/dog.

В журнале контейнера видно, что он начинает обслуживание сразу после запуска, но сообщает о готовности только после того, как загрузит из сети словарь размером примерно 8 Мбайт.

Чтобы развернуть сервис в Kubernetes, необходимо создать конфигурационный файл развертывания:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: dictionary-server
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: dictionary-server
    spec:
      containers:
      - name: server
        image: brendanburns/dictionary-server
        ports:
        - containerPort: 8080
        readinessProbe:
          httpGet:
            path: /ready
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 5
```

Чтобы создать на его основе реплицированный stateless-сервис, выполним следующую команду:

```
kubectl create -f dictionary-deploy.yaml
```

Теперь, когда у вас запущено несколько копий сервиса, нужен балансировщик нагрузки, который будет распределять запросы пользователей между ними. Балансировщик нагрузки не только равномерно распределяет нагрузку между экземплярами сервиса, но и предоставляет уровень абстракции для потребителей

реплицированного сервиса. Балансировщик нагрузки также предоставляет общее разрешимое сетевое имя, независимое от того, какой из экземпляров сервиса фактически обслужит запрос.

Балансировщик нагрузки в Kubernetes можно создать с помощью объекта `Service`:

```
kind: Service
apiVersion: v1
metadata:
  name: dictionary-server-service
spec:
  selector:
    app: dictionary-server
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

Сформировав конфигурационный файл, можно запустить сервис-словарь следующей командой:

```
kubectl create -f dictionary-service.yaml
```

Сервисы с закреплением сессий

В предыдущих примерах реализации паттерна реплицированного stateless-сервиса все пользовательские запросы направлялись всем экземплярам сервиса. Хотя такое решение гарантирует равномерное распределение нагрузки и отказоустойчивость, оно не всегда является предпочтительным. Часто лучше обеспечить направление запросов конкретного пользователя экземпляру сервиса, запущенному на определенной машине. Иногда это обусловлено кэшированием пользовательских данных в памяти: если запросы этого пользователя будут попадать на одну и ту же машину, то повысится коэффициент

попадания. Это также может быть обусловлено долгосрочной природой взаимодействия пользователя с системой, когда часть состояния хранится между запросами. Вне зависимости от причины можно адаптировать паттерн реплицированного stateless-сервиса к использованию в сервисах с закреплением сессий, гарантирующих, что все запросы конкретного пользователя будут адресованы одному и тому же экземпляру сервиса (рис. 5.3).

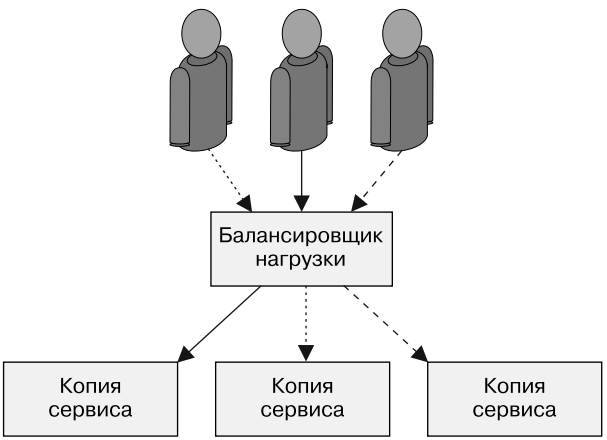

Рис. 5.3. Сервис с закреплением сессий, в котором все запросы конкретного пользователя адресуются одному и тому же экземпляру сервиса

Вообще говоря, закрепление сессий реализуется путем хеширования IP-адресов клиента и сервера и использования этого соответствия для определения того, какой экземпляр сервиса какой запрос будет обслуживать. Пока IP-адреса клиента и сервера остаются неизменными, все запросы этого клиента будут адресованы одному и тому же экземпляру сервиса.

Закрепление сессий часто реализуется на базе *консистентной хеш-функции*. Преимущества консистентной хеш-функции становятся очевидны при масштабировании сервиса в боль-

Закрепление сессий на основе IP-адресов работает в рамках кластера машин с локальными IP-адресами, но обычно плохо работает с внешними IP-адресами в силу трансляции адресов (NAT). Для закрепления внешних сессий предпочтительнее использовать закрепление на уровне приложения (например, посредством файлов cookie).

шую или меньшую сторону. Очевидно, что если количество копий сервиса меняется, то может поменяться и соответствие между конкретным пользователем и конкретным экземпляром сервиса. Консистентные хеш-функции минимизируют количество пользователей, у которых поменяется сопоставленный им экземпляр сервиса.

Сервисы с репликацией на уровне приложения

Во всех предыдущих примерах репликация и распределение нагрузки происходят на сетевом уровне сервиса. Распределение нагрузки не зависит от конкретного протокола взаимодействия узлов, находящегося в стеке над TCP/IP. Однако многие приложения общаются по протоколу HTTP, и, зная протокол общения узлов, можно расширить паттерн реплицированного stateless-сервиса дополнительной функциональностью.

Добавляем кэширующую прослойку

Иногда код stateless-сервиса достаточно сложен в вычислительном плане, несмотря на то что состояние сервиса не хранится.

Для обслуживания запросов может потребоваться обращение к базе данных, выполнение сложной обработки или визуализации данных. В таких условиях отнюдь не помешает добавить в приложение прослойку для кэширования. Кэш находится между

stateless-приложением и запросом конечного пользователя. Простейшая форма кэширования в веб-приложениях — применение кэширующего веб-прокси. Кэширующий прокси представляет собой просто-напросто HTTP-сервер, хранящий в памяти состояние запросов пользователей. Если два пользователя запросят одну и ту же веб-страницу, только один из запросов будет адресован приложению, второй будет обслужен из памяти кэша (рис. 5.4).

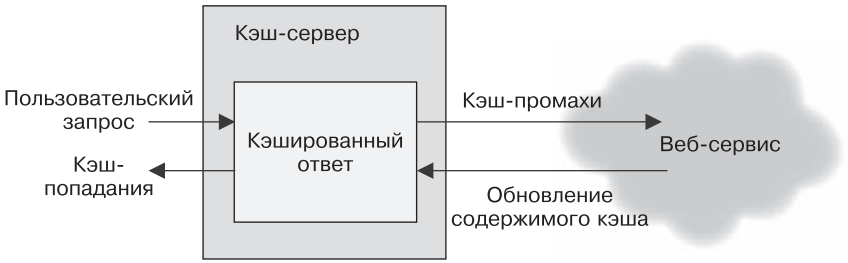

Рис. 5.4. Работа кэширующего сервера

Для наших целей воспользуемся кэширующим веб-сервером Varnish с открытым исходным кодом (https://varnish-cache.org/).

Развертывание кэширующего сервера

Простейший способ развертывания веб-кэша — рядом с каждым экземпляром сервиса при использовании паттерна Sidecar (рис. 5.5).

Такой подход при всей простоте имеет недостатки. В частности, вам придется масштабировать кэш одновременно с приложением. Это не всегда желательно. Для кэша следует использовать наименьшее количество экземпляров с наибольшим количеством памяти (например, не десять копий с 1 Гбайт памяти у каждой, а две копии с 5 Гбайт памяти у каждой). Для того чтобы понять, почему так лучше, представьте, что каждая страница кэшируется в каждом экземпляре. При десяти экземплярах кэша каждая

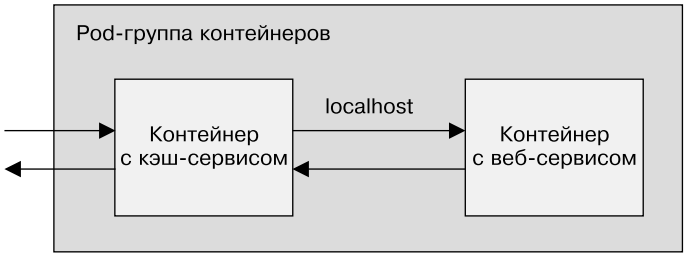

Рис. 5.5. Добавление кэширующего веб-сервера в виде контейнера-прицепа

страница будет записана десять раз, что уменьшит общее количество разных страниц, одновременно находящихся в кэше. Это снижает *коэффициент попадания* — долю запросов, обслуживаемых из кэша, что, в свою очередь, уменьшает полезность кэша.

И хотя желательно иметь как можно меньше крупных экземпляров кэш-серверов, небольших экземпляров веб-серверов должно быть как можно больше. Многие языки, например NodeJS, могут задействовать только одно процессорное ядро, и поэтому имеет смысл создавать много экземпляров сервиса, чтобы в полной мере использовать преимущества многоядерных систем, даже в рамках одной машины. Следовательно, имеет смысл настроить кэширующую прослойку как другой реплицированный stateless-сервис, находящийся над веб-сервисом (рис. 5.6).

Если не соблюдать осторожность, кэширование может нарушить механизм закрепления сессий. Причина в том, что, если вы используете привязку IP-адресов по умолчанию, все запросы будут отправляться с IP-адреса кэша, а не конечного пользователя сервиса. Если вы, следуя приведенному ранее совету, развернули небольшое количество кэш-серверов, привязка IP-адресов могла произойти таким образом, что некоторые экземпляры веб-сервиса не получают трафика. Вместо привязки сессии к IP следует использовать cookie-файлы или HTTP-заголовки.

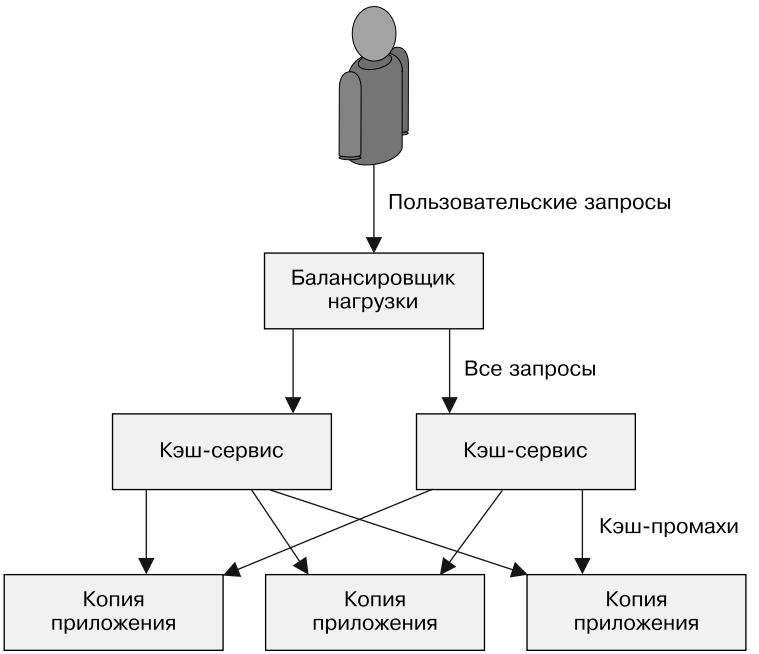

Рис. 5.6. Добавление кэширующей прослойки к реплицированному сервису

Практикум. Развертывание кэширующей прослойки

Сервис dictionary-server, который мы развернули ранее, распределяет трафик по экземплярам сервера-словаря и может быть найден по DNS-имени `dictionary-server-service`. Данный паттерн изображен на рис. 5.7.

Начнем создание кэширующей прослойки с настройки кэширующего сервера Varnish:

```
vcl 4.0;
backend default {
  .host = "dictionary-server-service";
  .port = "8080";
}
```

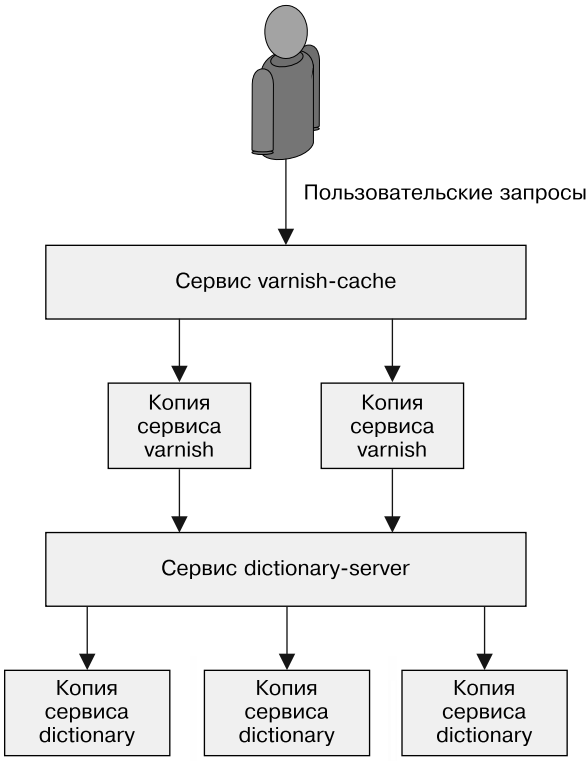

Рис. 5.7. Добавление кэширующей прослойки к серверу-словарю

Создадим объект `ConfigMap`, содержащий указанную конфигурацию:

```
kubectl create configmap varnish-config
--from-file=default.vcl
```

Теперь можно разворачивать реплицированный Varnish-кэш на основе следующего конфигурационного файла:

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

```
  name: varnish-cache
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: varnish-cache
    spec:
      containers:
      - name: cache
        resources:
          requests:
            # Зарезервируем 2 Гбайт памяти
            # для каждого экземпляра Varnish-кэша
            memory: 2Gi
            image: brendanburns/varnish
            command:
            - varnishd
            - -F
            - -f
            - /etc/varnish-config/default.vcl
            - -a
            - 0.0.0.0:8080
            - -s
            # Количество выделяемой здесь памяти должно
            # соответствовать количеству зарезервированной
            # памяти, указанному ранее
            - malloc,2G
            ports:
            - containerPort: 8080
            volumeMounts:
            - name: varnish
              mountPath: /etc/varnish-config
          volumes:
          - name: varnish
            configMap:
              name: varnish-config
```

Развернуть реплицированные Varnish-серверы можно следующей командой:

```
kubectl create -f varnish-deploy.yaml
```

Наконец, развернем балансировщик нагрузки для Varnish-кэша:

```
kind: Service
apiVersion: v1
metadata:
  name: varnish-service
spec:
  selector:
    app: varnish-cache
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Создать его можно с помощью такой команды:

```
kubectl create -f varnish-service.yaml
```

Расширение возможностей кэширующей прослойки

Теперь, когда мы развернули кэширующую прослойку для реплицированного stateless-сервиса, посмотрим, что еще он умеет делать, кроме собственно кэширования. Обратные HTTP-прокси вроде Varnish обычно имеют возможность расширения и, помимо кэширования, могут предоставлять дополнительные возможности.

Ограничение частоты запросов и защита от атак типа «отказ в обслуживании» (DoS)

Некоторые специалисты проектируют сайты с учетом защиты от DoS-атак. Все больше разработчиков сегодня проектируют программные интерфейсы. В связи с этим отказ в обслуживании может произойти из-за того, что разработчик некорректно настроил клиент, либо из-за того, что инженер, ответственный за доступность сайта, случайно запустил нагрузочные тесты

на рабочем сервере. Следовательно, имеет смысл добавить в кэширующую прослойку защиту от отказа в обслуживании, установив ограничение частоты запросов. Большинство обратных HTTP-прокси, например Varnish, поддерживают нечто похожее. В частности, у Varnish есть модуль throttle, который можно настроить так, чтобы он ограничивал частоту запросов с определенным путем с конкретных IP-адресов, в том числе для анонимных или зарегистрированных пользователей.

Если вы развертываете API, целесообразно иметь достаточно низкий лимит запросов для анонимных пользователей, который можно повысить после регистрации. Требуя авторизации, мы сможем проводить аудит, чтобы определить, чьи действия привели к неожиданно высокой нагрузке. Ограничение частоты запросов также служит барьером для потенциальных взломщиков, которым понадобится замаскироваться под нескольких пользователей, чтобы успешно реализовать атаку.

Когда количество запросов от одного пользователя достигает определенного лимита, сервер вернет ему ошибку с кодом 429, означающую превышение количества максимально допустимых запросов. Многим пользователям нужно будет знать, сколько еще они могут сделать запросов, прежде чем достигнут лимита. В связи с этим вам может понадобиться добавить в HTTP-заголовок информацию о количестве оставшихся запросов. Для подобной информации нет стандартного поля в HTTP-заголовке, однако многие API возвращают одну из разновидностей поля `X-RateLimit-Remaining`.

SSL-мост

Вдобавок к кэшированию с целью повышения производительности пограничный слой приложения также может выполнять функции SSL-моста. Даже если вы планируете использовать SSL для взаимодействия между внутренними слоями прило-

жения, вам все равно придется применять разные сертификаты для внешнего слоя и для взаимодействия внутренних сервисов. В самом деле, каждый внутренний сервис должен использовать свой собственный сертификат, чтобы можно было обеспечить независимое развертывание слоев.

К сожалению, Varnish нельзя применять для организации SSL-моста, но nginx имеет такую функциональность. Стало быть, в паттерне stateless-приложения нужен третий слой — он будет представлять собой реплицированный набор nginx-серверов, который обеспечит функцию SSL-моста для HTTPS-трафика и передаст его в расшифрованном виде кэширующему серверу Varnish. HTTP-трафик попадет в веб-кэш Varnish, который переадресует его веб-приложению (рис. 5.8).

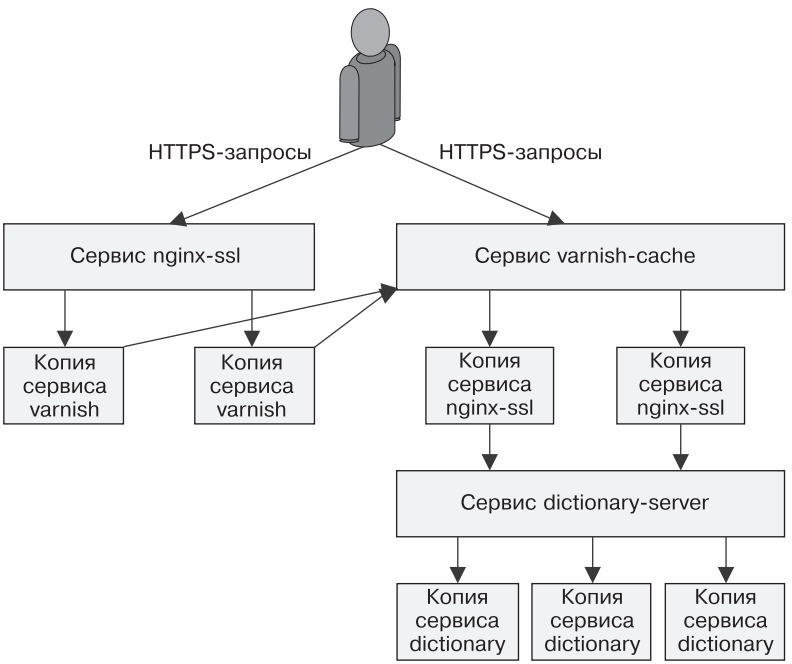

Рис. 5.8. Пример реплицированного stateless-сервиса

Практикум. Развертывание nginx и SSL-моста

Следующая инструкция описывает, как добавить SSL-мост на основе nginx к уже развернутому реплицированному сервису и кэшу.

Она подразумевает наличие у вас сертификата. Если вам нужно получить сертификат, то проще всего сделать это с помощью инструментов Let's Encrypt (https://letsencrypt.org/). Кроме того, для их создания можно воспользоваться инструментом openssl. Данная инструкция подразумевает, что файл сертификата носит имя server.crt, а файл секретного ключа — server.key. Самоподписанные сертификаты вызывают предупреждения безопасности во всех современных браузерах и никогда не должны использоваться в реальных системах.

Первый шаг — загрузить сертификат в Kubernetes:

```
kubectl create secret tls ssl --cert=server.crt --key=server.key
```

После загрузки сертификата в Kubernetes необходимо создать и настроить nginx для поддержки SSL:

```
events {
  worker_connections 1024;
}

http {
  server {
    listen 443 ssl;
    server_name my-domain.com www.my-domain.com;
    ssl on;
    ssl_certificate       /etc/certs/tls.crt;
    ssl_certificate_key   /etc/certs/tls.key;
    location / {
      proxy_pass http://varnish-service:80;
      proxy_set_header Host $host;
      proxy_set_header X-Forwarded-For
        $proxy_add_x_forwarded_for;
```

```
      proxy_set_header X-Forwarded-Proto $scheme;
      proxy_set_header X-Real-IP $remote_addr;
    }
  }
}
```

Как и в случае с Varnish, нужно преобразовать файл конфигурации в объект `ConfigMap` такой командой:

```
kubectl create configmap nginx-conf --from-file=nginx.conf
```

После загрузки сертификата и настройки nginx пришло время создать прослойку реплицированных stateless-серверов nginx:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-ssl
spec:
  replicas: 4
  template:
    metadata:
      labels:
        app: nginx-ssl
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
      - containerPort: 443
      volumeMounts:
      - name: conf
        mountPath: /etc/nginx
      - name: certs
        mountPath: /etc/certs
      volumes:
      - name: conf
        configMap:
          # Объект ConfigMap для nginx, созданный ранее
          name: nginx-conf
```

```
      - name: certs
        secret:
          # Ссылка на загруженные ранее сертификат
          # и секретный ключ
          secretName: ssl
```

Для создания реплицированных nginx-серверов нужно выполнить такую команду:

```
kubectl create -f nginx-deploy.yaml
```

Наконец, опубликуйте SSL-сервер nginx в виде сервиса:

```
kind: Service
apiVersion: v1
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx-ssl
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 443
      targetPort: 443
```

Чтобы создать сервис балансировщика, выполните команду:

```
kubectl create -f nginx-service.yaml
```

Если вы создали этот сервис в кластере Kubernetes, поддерживающем внешние балансировщики нагрузки, у вас появился открытый внешний сервис, принимающий запросы на внешний IP-адрес.

Чтобы узнать этот адрес, выполните команду:

```
kubectl get services
```

По этому адресу вы сможете обратиться к вашему сервису из браузера.

Резюме

Глава начиналась с описания простого паттерна для реплицированных stateless-сервисов. Затем мы дополнили его двумя реплицированными сервисами с балансировщиками нагрузки. Один выполняет функцию кэширования для повышения производительности, а другой — функцию SSL-моста для обеспечения защищенного соединения с клиентами. Полный паттерн реплицированного stateless-сервиса представлен на рис. 5.8.

Его можно развернуть в Kubernetes с помощью трех объектов развертывания и трех объектов — сервисов балансировщиков нагрузки. Полные исходные тексты примеров можно найти по адресу https://github.com/brendandburns/designing-distributed-systems.

6 Шардированные сервисы

В предыдущей главе мы обсудили значимость репликации stateless-сервисов для надежности, избыточности и масштабирования. В этой главе поговорим о шардированных сервисах. В рамках реплицированных сервисов, рассмотренных в предыдущей главе, каждая копия сервиса была равноценна и могла обслужить любой запрос. В отличие от реплицированных сервисов каждая *копия шардированного сервиса (шард)* может обслужить только часть запросов. *Узел балансировки нагрузки (корневой узел)* отвечает за изучение каждого запроса и перенаправление его соответствующему узлу (или узлам) для обработки. Разница между реплицированными и шардированными сервисами показана на рис. 6.1.

Репликация сервиса обычно используется для построения stateless-сервисов, а шардирование — для сервисов, хранящих состояние (stateful-сервисов). Необходимость шардинга данных возникает, когда объем данных становится слишком велик для обслуживания одной машиной. Шардинг позволяет масштабировать сервис в зависимости от объема обслуживаемых данных.

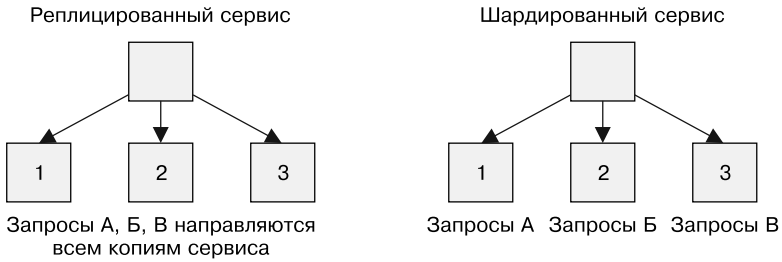

Рис. 6.1. Схемы реплицированного и шардированного сервисов

Шардирование кэша

Чтобы разобраться в структуре шардированной системы, нужно детально рассмотреть устройство *шардированного кэша*. Шардированный кэш — реализация кэша, стоящая между пользовательскими запросами и собственно распределенной реализацией кэша. Общая схема системы приведена на рис. 6.2.

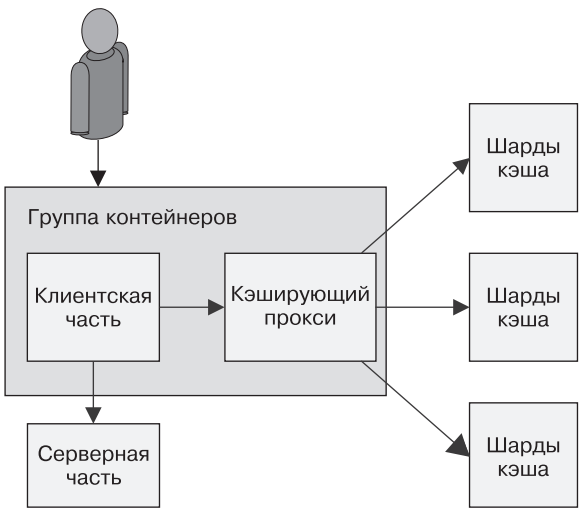

Рис. 6.2. Шардированный кэш

В главе 3 мы рассмотрели, как можно использовать паттерн Ambassador для распределения данных в шардированном сервисе. Здесь поговорим о том, как построить такой сервис. При проектировании шардированного кэша следует задать себе несколько вопросов:

- зачем нужен шардированный кэш;
- какова роль кэша в вашей архитектуре;
- нужен ли реплицированный и шардированный кэш;
- в чем состоит функция шардирования?

Зачем вам нужен шардированный кэш

Как уже упоминалось во введении, шардирование в первую очередь необходимо для увеличения объема хранимых в сервисе данных. Чтобы понять, как это помогает кэшированию, рассмотрим следующую систему. В каждом экземпляре кэша есть 10 Гбайт памяти для хранения результатов. Каждый экземпляр кэша может обслуживать до 100 запросов в секунду (RPS). Допустим, в нашем сервисе хранится 200 Гбайт данных, а ожидаемая нагрузка составляет 1000 RPS. Очевидно, требуется десять экземпляров кэша, чтобы удовлетворить 1000 запросов в секунду (десять экземпляров по 100 RPS на экземпляр). Проще всего будет развернуть этот сервис в реплицированном виде, как показано в предыдущей главе. Но, если развернуть его таким образом, распределенный кэш сможет хранить не более 5 % (10 из 200 Гбайт) общего набора данных. Так происходит потому, что каждый экземпляр кэша независим от остальных, а значит, хранит примерно те же данные, что и остальные. Это отличный подход для обеспечения избыточности, который совершенно не способствует эффективному использованию памяти. Если же мы развернем шардированный на десять частей кэш, то все так же сможем обслуживать нужное количество запросов в секунду

(10 × 100 все еще равно 1000). Однако, поскольку каждый экземпляр кэша работает со своей отдельной частью данных, мы можем хранить там 50 % данных (10 × 10 из 200 Гбайт). Десятикратное увеличение объема кэшируемых данных означает, что кэш-память используется гораздо более эффективно, поскольку каждый элемент данных попадает только в один кэш.

Роль кэша в производительности системы

В главе 5 мы обсудили, как использовать кэш с целью оптимизации производительности для конечного пользователя и сокращения задержек. Не была, однако, рассмотрена роль кэша в производительности, надежности и стабильности приложения.

Проще говоря, важно задать себе следующий вопрос: если кэш откажет, как это повлияет на ваших пользователей и на работу сервиса в целом?

Когда мы обсуждали реплицированный кэш, этот вопрос был менее актуален, так как кэш масштабировался горизонтально, то есть отказ одного из экземпляров приводил бы только к кратковременным неисправностям. Аналогичным образом рассмотренный кэш поддерживал масштабирование в связи с выросшей нагрузкой, не влияя при этом на конечных пользователей.

В случае с шардированным кэшем все оказывается несколько иначе. Поскольку конкретный пользователь или запрос всегда соответствует одному и тому же шарду, в случае его отказа кэш-промахи будут происходить до тех пор, пока шард не будет восстановлен. Учитывая временность нахождения данных в кэше, такие кэш-промахи не являются проблемой сами по себе — система должна знать, где взять данные. Однако извлечение данных в отсутствие кэша происходит намного медленнее, что означает снижение производительности для конечных пользователей.

Производительность кэша выражается в виде *коэффициента попадания запросов*. Коэффициент попадания — доля запросов, ответ на которые содержится в кэше. В конечном итоге коэффициент попадания характеризует общую максимальную нагрузку на распределенную систему и влияет на производительность и мощность системы в целом.

Представьте, что уровень обработки запросов вашего приложения поддерживает обработку 1000 запросов в секунду.

При превышении этого показателя система начинает возвращать HTTP-ошибки с кодом 500. Если вы добавите кэш с 50%-ной вероятностью попадания, количество обрабатываемых запросов возрастет до 2000 в секунду. Так происходит потому, что из 2000 запросов одна половина может быть обслужена кэшем, а другая — уровнем обработки запросов. В данном примере кэш довольно критичен для работы сервиса, поскольку в случае его отказа уровень обработки запросов окажется перегружен и половина запросов завершится ошибкой. Именно поэтому имеет смысл оценить емкость сервиса в 1500 запросов в секунду, а не в полные 2000. Это позволит удержать сервис в стабильном состоянии даже при отказе половины экземпляров кэша.

Производительность системы, однако, не ограничивается количеством обрабатываемых в единицу времени запросов. Производительность, с точки зрения конечного пользователя, также определяется *задержкой* выполнения запросов. Получить результат из кэша, как правило, гораздо быстрее, чем сформировать его с нуля. Следовательно, кэш повышает не только количество одновременно обрабатываемых запросов, но и скорость их обработки. Почему? Представьте, что система обслуживает запрос пользователя за 100 миллисекунд. Добавим кэш с вероятностью попадания 25 %, который возвращает результат за 10 миллисекунд. Средняя задержка обработки запроса, таким

образом, уменьшилась до 77,5 миллисекунд. Кэш не только увеличивает количество обрабатываемых в секунду запросов, но и ускоряет выполнение каждого отдельного запроса, поэтому замедление обработки запросов в результате отказа части экземпляров кэша или развертывания новой его версии беспокоит нас не слишком сильно. Однако в некоторых случаях влияние на производительность окажется настолько велико, что запросы начнут скапливаться в очередях и часть из них будет отклоняться по истечении времени ожидания. Никогда не будет лишним выполнять нагрузочное тестирование сервиса как при наличии, так и при отсутствии кэша, чтобы понять его влияние на общую производительность системы.

Наконец, нужно думать не только об отказах. Если вы хотите обновить или повторно развернуть шардированный кэш, не получится просто развернуть новую копию сервиса и рассчитывать на то, что он сразу же возьмет на себя нагрузку. Развертывание новой версии шардированного кэша в общем случае приведет к временной потере производительности. Другим, более сложным решением будет репликация шардов.

Реплицированный и шардированный кэш

Иногда система оказывается настолько зависимой от кэша в плане задержек или нагрузки, что потеря даже одного шарда в результате отказа или в процессе обновления оказывается неприемлемой.

Или же нагрузка на определенный шард становится слишком велика и вам приходится его масштабировать, чтобы он выдерживал объем работ. Указанные причины могут подтолкнуть вас к развертыванию одновременно реплицированного и шардированного сервиса. Шардированный сервис с репликацией совмещает паттерн построения реплицированного сервиса, рассмотренный в предыдущей главе, с паттерном шардирования, описанным

в предыдущих разделах. По сути, каждый шард кэша в таком случае реализуется не одним сервером, а реплицированным сервисом.

Такая архитектура более сложна в реализации и развертывании, но имеет определенные преимущества перед просто шардированным сервисом. Что наиболее важно, замена одиночного сервера реплицированным сервисом повышает устойчивость шардов к отказам и обеспечивает их доступность в случае отказа одной из реплик. Вместо того чтобы закладывать в систему устойчивость к снижению производительности из-за отказа шардов, можно рассчитывать на улучшение производительности за счет использования кэша. Если вы готовы обеспечить избыточный резерв вычислительной мощности для шардов, развертывание новой версии даже в периоды пиковой нагрузки для вас не представляет никакой опасности. Теперь нет необходимости ждать, когда нагрузка спадет.

Кроме того, поскольку каждый шард представляет собой независимый реплицированный сервис, любой шард можно масштабировать в зависимости от его текущей нагрузки. Такого рода «горячее» шардирование мы рассмотрим в конце этой главы.

Практикум. Развертывание реализации паттерна Ambassador и сервиса memcache для организации шардированного кэша

В главе 3 мы рассмотрели процесс развертывания шардированного сервиса Redis. Развертывание шардированного сервиса memcache происходит подобным образом.

Сначала развернем memcache с помощью Kubernetes-объекта `StatefulSet`:

```
apiVersion: apps/v1beta1
kind: StatefulSet
```

```
metadata:
  name: sharded-memcache
spec:
  serviceName: "memcache"
  replicas: 3
  template:
    metadata:
      labels:
        app: memcache
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: memcache
        image: memcached
        ports:
        - containerPort: 11211
          name: memcache
```

Сохраните этот код в файле с именем `memcached-shards.yaml`, который можно развернуть командой `kubectl create -f memcached-shards.yaml`. В результате этого будут созданы три контейнера с запущенным сервисом memcached.

Как и в примере с Redis, нам также необходимо создать Kubernetes-сервис, который назначит DNS-имена созданным экземплярам memcached. Он будет выглядеть следующим образом:

```
apiVersion: v1
kind: Service
metadata:
  name: memcache
  labels:
    app: memcache
spec:
  ports:
  - port: 11211
    name: memcache
  clusterIP: None
  selector:
    app: memcache
```

Сохраните этот код в файле с именем `memcached-service.yaml` и разверните полученный файл командой `kubectl create -f memcached-shards.yaml`. Теперь на вашем DNS-сервере должны появиться записи для хостов `memcache-0.memcache`, `memcache-1.memcache` и т. д. Как и в случае с redis, эти имена следует использовать для настройки twemproxy (https://github.com/twitter/twemproxy).

```
memcache:
  listen: 127.0.0.1:11211
  hash: fnv1a_64
  distribution: ketama
  auto_eject_hosts: true
  timeout: 400
  server_retry_timeout: 2000
  server_failure_limit: 1
  servers:
    - memcache-0.memcache:11211:1
    - memcache-1.memcache:11211:1
    - memcache-2.memcache:11211:1
```

Из конфигурационного файла видно, что запросы к memcache обслуживаются по адресу localhost:6379, так что контейнер приложения может получить доступ к контейнеру-послу. Теперь можно развернуть его в pod-группу к контейнеру-послу с помощью Kubernetes-объекта `ConfigMap`, который можно создать командой `kubectl create configmap --from-file=nutcracker.yaml twem-config`.

Подготовительные действия завершены, и теперь вы можете развернуть пример реализации паттерна Ambassador.

Определите pod-группу контейнеров, которая выглядит следующим образом:

```
apiVersion: v1
kind: Pod
metadata:
  name: sharded-memcache-ambassador
spec:
```

```
  containers:
    # Сюда необходимо подставить имя контейнера приложения,
    # например:
    # - name: nginx
    # image: nginx
    # Здесь указываем имя контейнера-посла
    - name: twemproxy
      image: ganomede/twemproxy
      command:
      - nutcracker
      - -c       - /etc/config/nutcracker.yaml
      - -v       - 7
      - -s       - 6222
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: twem-config
```

Сохраните этот текст в файл с именем `memcached-ambassador-pod.yaml`, затем разверните его такой командой:

```
kubectl create -f memcached-ambassador-pod.yaml
```

Если не хотите использовать паттерн Ambassador, можно его не использовать. Вместо этого можно развернуть реплицированный *сервис маршрутизации запросов шардам*. У паттерна Ambassador, по сравнению с этим сервисом, есть свои достоинства и недостатки. Ценность сервиса маршрутизации состоит в снижении сложности. Вам не придется разворачивать контейнер-посол в каждой pod-группе, которой нужен доступ к шардированному сервису memcache. Доступ к нему может быть получен с помощью именованного сервиса с балансировкой нагрузки. У разделяемого сервиса есть и два недостатка. Во-первых, поскольку сервис используется совместно, его придется масштабировать по мере увеличения нагрузки. Во-вторых, разделяемый сервис — дополнительный участок пути сетевого маршрута, вносящий лишнюю

задержку в обработку запросов и занимающий часть пропускной способности распределенной системы.

Чтобы развернуть разделяемый сервис маршрутизации запросов, нужно немного изменить конфигурацию twemproxy, чтобы он принимал запросы во всех интерфейсах, а не только в петлевом:

```
memcache:
  listen: 0.0.0.0:11211
  hash: fnv1a_64
  distribution: ketama
  auto_eject_hosts: true
  timeout: 400
  server_retry_timeout: 2000
  server_failure_limit: 1
  servers:
    - memcache-0.memcache:11211:1
    - memcache-1.memcache:11211:1
    - memcache-2.memcache:11211:1
```

Сохраните этот код в файл с именем `shared-nutcracker.yaml`, затем создайте соответствующий объект `ConfigMap` с помощью команды `kubectl`:

```
kubectl create configmap --from-file=shared-nutcracker.yaml
shared-twem-config
```

Разверните реплицированный сервис маршрутизации запросов шардам:

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

```
      containers:
      - name: twemproxy
        image: ganomede/twemproxy
        command:
        - nutcracker
        - -c
        - /etc/config/shared-nutcracker.yaml
        - -v
        - 7
        - -s
        - 6222
        volumeMounts:
        - name: config-volume
          mountPath: /etc/config
      volumes:
      - name: config-volume
        configMap:
          name: shared-twem-config
```

Сохраните код в файл с именем `shared-twemproxy-deploy.yaml`, затем разверните реплицированный маршрутизатор запросов шардам с помощью `kubectl`:

```
kubectl create -f shared-twemproxy-deploy.yaml
```

Чтобы завершить развертывание маршрутизатора запросов шардам, необходимо создать балансировщик нагрузки, обрабатывающий запросы:

```
kind: Service
apiVersion: v1
metadata:
  name: shard-router-service
spec:
  selector:
    app: shared-twemproxy
  ports:
    - protocol: TCP
      port: 11211
      targetPort: 11211
```

Балансировщик нагрузки разворачивается командой `kubectl create -f shard-router-service.yaml`.

Шардирующие функции

На данный момент мы обсудили процесс проектирования и развертывания просто шардированного и реплицированного шардированного кэша. Настало время уделить внимание тому, каким образом трафик переадресуется тому или иному шарду. Рассмотрим шардированный сервис с десятью независимыми шардами. Дан пользовательский запрос *Req*. Как выяснить, какой из шардов S с номерами от 0 до 9 должен обработать запрос? Задача *шардирующей функции* — определить данное соответствие. Шардирующие функции похожи на хеш-функции, с которыми вы наверняка уже сталкивались, например, при изучении ассоциативных массивов. Действительно, хеш-таблицу в виде массива цепочек можно считать примером шардированного сервиса. Для заданных *Req* и *Shard* шардирующая функция должна установить между ними соответствие вида:

$$Shard = ShardingFunction(Req).$$

Шардирующая функция часто реализуется с использованием *хеш-функции* и оператора взятия остатка от деления (%). Хеш-функции преобразуют произвольные цепочки байтов в целые числа *фиксированной длины*. Хеш-функция имеет две важные с точки зрения шардинга характеристики.

- *Детерминированность* — одинаковые цепочки байтов на входе должны порождать одинаковый результат на выходе.
- *Равномерность* — значения хеш-функции должны иметь равномерное распределение.

Для шардированного сервиса первостепенное значение имеют детерминированность и равномерность хеш-функции. Детерми-

нированность важна, так как обеспечивает то, что определенный запрос *R* всегда будет попадать на один и тот же шард сервиса. Равномерность хеш-функции обеспечивает равномерное распределение нагрузки между шардами.

К счастью, библиотеки современных языков программирования включают целый ряд качественных хеш-функций. Диапазон значений этих функций зачастую существенно превышает количество шардов в системе. Чтобы сузить этот диапазон, необходимо воспользоваться оператором взятия остатка от деления. Возвращаясь к нашему сервису из десяти шардов, нетрудно догадаться, что функция шардирования будет иметь следующий вид:

$$Shard = hash(Req) \ / \ 10.$$

Оператор взятия остатка от деления не нарушает свойств детерминированности и равномерности хеш-функции.

Выбор ключа

Может показаться заманчивым просто взять встроенную хеш-функцию языка программирования и применить ее ко всему объекту. Однако получившаяся в результате этого функция шардирования не будет идеальной.

Чтобы лучше разобраться в этом, рассмотрим запрос с тремя полями:

- время запроса;
- IP-адрес клиента;
- путь HTTP-запроса (например, `/some/page.html`).

Очевидно, что при простом хешировании запроса целиком запросы `{12:00, 1.2.3.4, /some/file.html}` и `{12:01, 5.6.7.8, /some/file.html}` будут соответствовать разным шардам.

Шардирующая функция выдает разные результаты, поскольку IP-адреса и временные метки запросов не совпадают. Но в большинстве случаев ответ на HTTP-запрос не зависит ни от адреса клиента, ни от временной метки запроса. Следовательно, намного лучше использовать шардирующую функцию вида `shard(request.path)`, а не хешировать весь запрос. Если в качестве ключа шардирования использовать путь запроса, то оба запроса попадут на один и тот же шард и второй запрос уже будет обслужен из кэша.

Иногда IP-адрес важен для запросов, возвращаемых интерфейсной частью. IP-адрес клиента может, скажем, применяться для определения страны, в которой находится пользователь. Это позволяет возвращать разный контент (например, на разных языках) разным пользователям. В таких случаях применение шардирующей функции от HTTP-пути может приводить к ошибкам — запрос с французского IP-адреса может быть обслужен из англоязычного кэша. Такая шардирующая функция будет слишком *общей*, поскольку группирует запросы с неидентичными ответами.

Ставить номер шарда в зависимость одновременно от IP-адреса и HTTP-пути — тоже не лучший подход.

Два запроса с разных французских IP-адресов могут попасть на разные шарды, что снижает эффективность шардирования. Такая функция шардирования слишком *специфична*, поскольку может распределять запросы с идентичными ответами в разные группы. В таком случае лучше использовать следующую функцию шардирования:

```
shard ( country ( request.ip ), request.path )
```

Она сначала определяет страну по IP-адресу, затем использует ее в качестве части ключа. Запросы из Франции будут адресованы одному шарду, а запросы из Америки — другому.

При проектировании шардированной системы критически важно корректно определить ключ шардирования. Для этого нужно хорошо понимать, какие запросы могут быть адресованы вашей системе.

Консистентные хеш-функции

Первоначальная настройка шардов в новой распределенной системе довольно проста — достаточно настроить соответствующие шарды и шардированные сервисы. А что случится, если вы захотите изменить количество шардов в шардированной системе? Повторное шардирование — часто довольно затратный процесс.

Чтобы разобраться, почему это так, вернемся к ранее рассмотренному примеру с шардированным кэшем. Оркестратор контейнеров позволит без труда увеличить кэш с 10 до 11 экземпляров. Но каков будет эффект от изменения шардирующей функции с `hash(Req) % 10` на `hash(Req) % 11`? Когда вы примените новую шардирующую функцию, значительная часть запросов уйдет на другие шарды, нежели те, что были назначены предыдущей функцией. Это существенно увеличит *процент кэш-промахов* в шардированном кэше до тех пор, пока шарды не заполнятся ответами на вновь сопоставленные с ними запросы. Применение новой функции шардирования к шардированному кэшу в худшем случае приведет к кэш-промахам в 100 % вариантов.

Чтобы решить подобную проблему, многие шардирующие функции прибегают к использованию *консистентных хеш-функций*. Эти хеш-функции устроены таким образом, что при количестве ключей K и увеличении количества шардов до N гарантированно окажется перенаправлено не более K / N запросов. Например, при использовании консистентной хеш-функции для шардирования

кэша из рассматриваемого примера переход с 10 шардов на 11 вызовет перенаправление менее 10 % запросов ($K / 11$). Это гораздо лучше, чем потерять весь шардированный сервис.

Практикум. Построение консистентного шардированного прокси-сервера

Первый вопрос, которым стоит задаться при шардировании HTTP-запросов, — какой ключ использовать в шардирующей функции? Есть несколько подходов, но в общем случае неплохо подойдет путь HTTP-запроса, совмещенный с параметрами запроса и всем тем, что делает запрос уникальным. И это *без учета* cookies и языка/страны (например, en-us). Если ваш сервис позволяет пользователю выполнять тонкую настройку параметров, их значения также стоит сделать частью ключа.

В качестве шардирующего прокси может выступать nginx.

```
worker_processes 5;
error_log  error.log;
pid        nginx.pid;
worker_rlimit_nofile 8192;

events {
  worker_connections 1024;
}

http {
  # Определяем именованный «обработчик», который можно будет
  # указать в директиве proxy ниже
  upstream backend {
    # Хешируем URI-адрес запроса с использованием
    # консистентной хеш-функции
    hash $request_uri consistent
    server web-shard-1.web;
    server web-shard-2.web;
```

```
    server web-shard-3.web;
  }

  server {
    listen localhost:80;
    location / {
      proxy_pass http://backend;
    }
  }
}
```

Обратите внимание, что в качестве ключа мы используем полный URI запроса, а также указываем ключевое слово `consistent`, чтобы nginx задействовал консистентную хеш-функцию.

Шардирование реплицированных сервисов

В большей части примеров в этой главе описывается шардирование кэша. Кэш, безусловно, не единственный сервис, которому шардирование пойдет на пользу. Шардирование подойдет для любого сервиса, в котором хранится больше данных, чем может поместиться на одной машине. В отличие от предыдущих примеров, ключ и функция шардирования являются не частью HTTP-запроса, а частью пользовательского контекста.

Рассмотрим, например, реализацию крупномасштабной многопользовательской игры. Мир такой игры, скорее всего, слишком велик для хранения на одной машине. Маловероятно, что игроки, находящиеся в этом мире далеко друг от друга, будут как-то взаимодействовать. Следовательно, игровой мир может быть *шардирован* на несколько машин. Ключом шардирующей функции в этом случае будет местоположение пользователя на игровой карте. Таким образом, все игроки, находящиеся

в одной части карты, будут обслуживаться одной и той же группой серверов.

Системы с «горячим» шардированием

В идеале нагрузка на шардированный кэш должна быть равномерной, но во многих случаях это не так. За счет этого появляются «горячие» шарды, поскольку естественные закономерности в распределении нагрузки приводят к тому, что на одни шарды приходится больше трафика, чем на другие.

Рассмотрим, например, шардированный кэш пользовательских фотографий. Когда какая-то фотография приобретает «вирусную» популярность, на нее приходится несоразмерно больше трафика, чем на другие, и шард кэша, содержащий эту фотографию, становится «горячим». Когда в реплицированном шардированном кэше возникает такая ситуация, этот шард можно масштабировать в соответствии с выросшей нагрузкой.

Если для шардов настроено автоматическое масштабирование, можно в зависимости от роста или падения нагрузки динамически выделять и освобождать ресурсы, предназначенные для реплицированных шардов. Данный процесс проиллюстрирован на рис. 6.3.

Сначала все три шарда обрабатывают одинаковое количество трафика. Затем трафик перераспределяется таким образом, что на шард А приходится в четыре раза больше трафика, чем на шарды Б и В. Система с «горячим» шардированием перемещает шард Б на машину с шардом В, а шард А реплицирует на вторую машину. Реплики теперь делят трафик поровну.

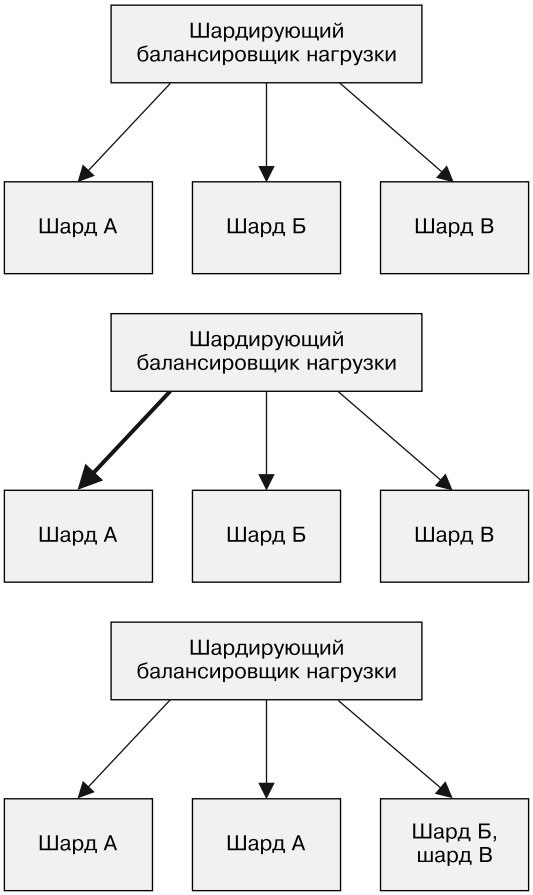

Рис. 6.3. Пример системы с «горячим» шардированием. Сначала шарды нагружены равномерно, но когда нагрузка на шард А увеличивается, он реплицируется на две машины, а шарды Б и В попадают на одну

7 Паттерн Scatter/Gather

До сих пор мы изучали реплицированные системы, которые масштабируются по количеству обрабатываемых в секунду запросов (реплицированные сервисы без внутреннего состояния) либо по объему обрабатываемых данных (шардированные данные). В этой главе описывается паттерн *Scatter/Gather*, в котором репликация используется для масштабирования по времени. В частности, паттерн Scatter/Gather позволяет добиться параллелизма в обработке запросов, за счет чего вы сможете обслуживать их намного быстрее, чем при последовательной обработке.

Подобно паттернам реплицированных и шардированных систем, паттерн Scatter/Gather — древовидный паттерн, в котором корневой узел распределяет запросы, а терминальные узлы их обрабатывают. Однако, в отличие от реплицированных и шардированных систем, запросы в Scatter/Gather-системах распределяются между всеми репликами сервиса. Каждая реплика делает небольшую часть работы и возвращает результат кор-

невому узлу. Корневой узел затем собирает частичные ответы в один общий ответ, который и возвращается клиенту. Паттерн Scatter/Gather схематически изображен на рис. 7.1.

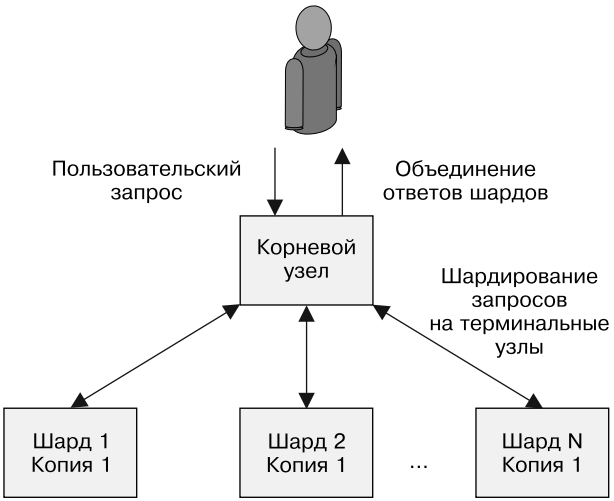

Рис. 7.1. Паттерн Scatter/Gather

Он весьма полезен, если обработка запроса подразумевает большое количество независимых действий. Паттерн Scatter/Gather может рассматриваться как шардирование вычислений, необходимых для обработки запроса, в противовес шардированию данных (шардирование данных может быть частью этого процесса).

Scatter/Gather с распределением нагрузки корневым узлом

В простейшем варианте паттерна Scatter/Gather все терминальные узлы идентичны, а работа распределяется между ними для ускорения обработки запроса. Этот паттерн напоминает

решение «чрезвычайно параллельной» задачи. Задачу можно разбить на множество мелких фрагментов, результаты решения которых можно склеить, чтобы получить полный результат.

Разберем его принцип работы на конкретном примере. Представьте, что вам нужно обслужить запрос *R*, который на одном ядре выполняется за одну минуту и выдает ответ A. При создании многопоточного приложения мы можем распараллелить обработку запроса на несколько ядер. На 30-ядерном процессоре (обычно ядер 32, но для ровного счета возьмем 30) время обработки запроса снижается до 2 секунд (60 секунд машинного времени, разделенные на 30 потоков, дают 2 секунды). Но даже 2 секунды — это достаточно долго. Достижение полного параллелизма в рамках одного процесса практически невозможно, поскольку пропускная способность памяти, сетевого подключения или диска становится узким местом. Вместо распараллеливания приложения на несколько ядер одной машины можно использовать паттерн Scatter/Gather, чтобы распараллелить запросы на несколько процессов, работающих на нескольких машинах. Таким образом, мы можем сократить среднее время обработки запросов, поскольку нас перестает ограничивать количество ядер процессора на одной машине.

Узким местом остается процессор, поскольку пропускная способность памяти, сетевого интерфейса и жесткого диска распределена на несколько машин. Кроме того, поскольку каждая машина в дереве Scatter/Gather способна обработать все запросы, корневой узел дерева может динамически распределять нагрузку между узлами в зависимости от времени их реакции. Если по какой-то причине некоторый терминальный узел отвечает медленнее остальных (например, на его ресурсы посягает жадный процесс-сосед), то корневой узел может динамически перераспределить нагрузку, чтобы обеспечить необходимую скорость реакции.

Практикум. Распределенный поиск в документах

Рассмотрим паттерн Scatter/Gather в действии на примере задачи поиска всех документов, содержащих слова «кот» и «собака», в большой базе документов. Можно открывать все документы подряд и искать в них соответствия образцу поиска, затем возвращать пользователю набор документов, в которых есть оба слова.

Как вы можете себе представить, этот процесс довольно длителен, поскольку для каждого запроса необходимо открывать и считывать большое количество файлов. Чтобы ускорить обработку, документы можно *проиндексировать*. Индекс, по сути, представляет собой ассоциативный массив, ключами в котором выступают отдельные слова, а значениями — списки документов, содержащих данное слово.

Теперь вместо того, чтобы искать совпадения во всех документах, совпадение отдельного слова можно найти простым ассоциативным поиском. Правда, за счет этого теряется одна важная возможность. Помните, что мы ищем все документы, в которых есть слова «кот» и «собака». Поскольку индекс содержит только единичные слова, а не наборы слов, все равно придется искать документы, включающие оба слова. К счастью, их множество представляет собой пересечение множеств документов, содержащих отдельные слова.

Такой подход позволяет использовать поиск по документам в качестве примера реализации паттерна Scatter/Gather. Когда корневой узел получает поисковый запрос, он выделяет в нем термы и распределяет его выполнение на две машины (одной — слово «кот», а второй — «собака»). Каждая из машин возвращает список документов, соответствующих ее поисковому терму, а корневой узел возвращает список документов, содержащих оба терма.

Схематически процесс показан на рис. 7.2: терминальный узел, искавший слово «кот», возвращает множество `{doc1, doc2, doc4}`, а узел, искавший слово «собака», возвращает множество `{doc1, doc3, doc4}`. Корневой узел затем ищет пересечение этих множеств и возвращает пользователю множество `{doc1, doc4}`.

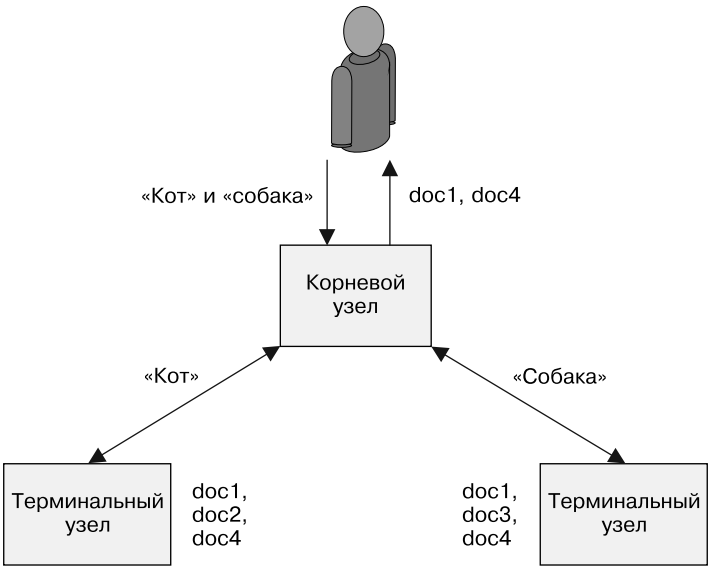

Рис. 7.2. Пример поисковой системы Scatter/Gather с шардированием по термам поискового запроса

Scatter/Gather с шардированием терминальных узлов

Хотя при применении реплицированного варианта паттерна Scatter/Gather сокращается время обработки пользовательских запросов, он не позволит масштабировать сервис сверх объема данных, который можно хранить в памяти или на диске одной

машины. Как и в случае с ранее рассмотренным примером реплицированного сервиса, нетрудно построить реплицированную Scatter/Gather-систему. Но по достижении определенного объема данных становится необходимым вводить шардирование, чтобы иметь возможность обрабатывать больше данных, чем может уместиться на одной машине.

Ранее, когда шардирование вводилось для масштабирования реплицированных систем, оно осуществлялось на уровне отдельных запросов. Для определения того, на какой узел направить запрос, использовался некоторый его фрагмент. Этот узел впоследствии полностью обрабатывал запрос и выдавал ответ пользователю. В случае шардирования Scatter/Gather, напротив, запрос отправляется всем терминальным узлам (шардам) в системе. Каждый терминальный узел обрабатывает запрос, используя данные, содержащиеся в своем шарде. Частичный ответ возвращается корневому узлу — он объединяет все частичные ответы в один полный ответ, который и возвращается пользователю.

В качестве конкретного примера такой архитектуры рассмотрим реализацию поиска в большом массиве документов (например, среди всех патентов в мире). Такой массив данных слишком велик для одной машины, поэтому данные шардируются среди нескольких экземпляров сервиса. Первые 100 тысяч патентов, к примеру, могут находиться на первой машине, вторые 100 тысяч — на второй и т. д. Заметьте, подобная схема шардирования плоха тем, что по мере регистрации новых патентов придется добавлять в систему новые шарды. На практике следует использовать остаток от деления номера патента на общее количество шардов. Когда пользователь делает поисковый запрос по всем патентам с определенным словом (к примеру, «ракеты»), запрос отправляется всем экземплярам сервиса, каждый из которых затем выполняет поиск совпадений в своем шарде массива

патентов. В ответ на шардированный запрос все найденные совпадения возвращаются корневому узлу сервиса. Корневой узел затем объединяет все ответы в один общий ответ, который содержит все патенты, включающие определенное слово. Схема работы такого поисковика изображена на рис. 7.3.

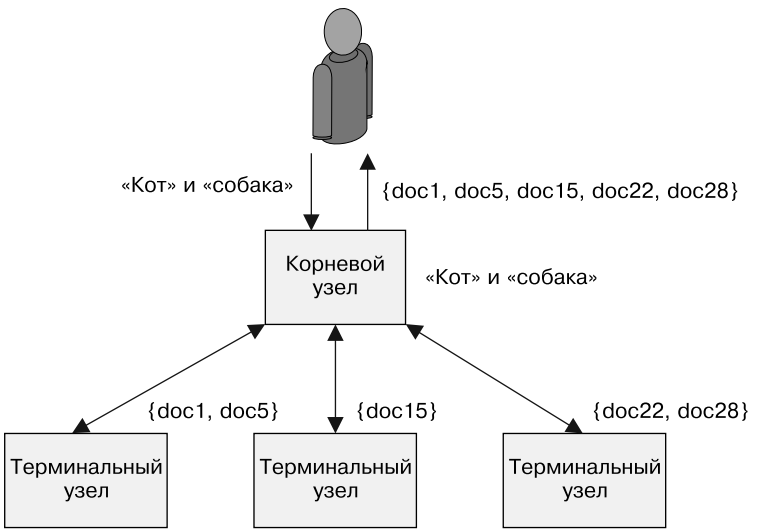

Рис. 7.3. Выполнение запроса с конъюнкцией в поисковой системе типа Scatter/Gather

Практикум. Шардированный поиск в документах

В предыдущем примере подзапросы отдельных термов распределялись по всему кластеру. Это работает только в том случае, когда все документы реплицированы на все машины в Scatter/Gather-дереве. Если в каждом отдельном терминальном узле дерева не хватает места для хранения всех документов, приходится прибегать к шардированию, чтобы разные подмножества документов хранились в разных узлах.

Это значит, что, когда пользователь запрашивает все документы, содержащие слова «кот» и «собака», его запрос отправляется всем терминальным узлам Scatter/Gather-дерева. Каждый терминальный узел возвращает набор известных ему документов, содержащих слова «кот» и «собака». Ранее корневой узел отвечал за пересечение наборов документов, возвращенных для каждого поискового терма. В случае шардированного сервиса корневой узел отвечает за вычисление теоретико-множественного объединения наборов документов, возвращенных шардами, которое и является конечным результатом, отправляемым пользователю.

Еще раз посмотрите на рис. 7.3. Первый терминальный узел обслуживает документы 1–10 и возвращает `{doc1, doc5}`.

Второй терминальный узел обслуживает документы 11–20 и возвращает `{doc15}`. Третий терминальный узел обслуживает документы 21–30 и возвращает `{doc22, doc28}`. Корневой узел объединяет ответы и возвращает множество `{doc1, doc5, doc15, doc22, doc28}`.

Выбор подходящего количества терминальных узлов

Может показаться, что в рамках паттерна Scatter/Gather всегда имеет смысл реплицировать вычисления на как можно большее количество узлов. Распараллеливая вычисления, вы сокращаете время обработки конкретного запроса. Увеличение степени распараллеливания несет с собой дополнительные расходы, поэтому для достижения максимальной производительности в распределенной системе чрезвычайно важно правильно выбрать количество терминальных узлов.

Чтобы понять, почему так происходит, следует рассмотреть две вещи. Во-первых, обработка каждого конкретного запроса

подразумевает определенные накладные расходы. Требуется время на анализ запроса, на его пересылку по сети и т. д. В общем случае накладные расходы на обработку запроса операционной системой постоянны и сравнительно невелики по отношению ко времени обработки запроса в пользовательском режиме. Соответственно, при оценке производительности реализации паттерна Scatter/Gather ими, как правило, можно пренебречь.

Важно, однако, понимать, что объем накладных расходов в реализации паттерна Scatter/Gather растет с увеличением количества терминальных узлов. Поэтому, несмотря на их небольшой объем, с ростом степени распараллеливания расходы на них со временем могут превысить расходы на реализацию бизнес-логики приложения. А это значит, что рост производительности за счет распараллеливания носит асимптотический характер.

Помимо того что добавление терминальных узлов может и не ускорить обработку запросов, Scatter/Gather-системы также подвержены «эффекту отстающего». Чтобы понять причину этого, важно помнить, что в Scatter/Gather-системе корневой узел сможет отправить ответ конечному пользователю не ранее, чем дождется ответа от *всех* терминальных узлов. Поскольку необходимо получить данные от всех узлов, общее время обработки запроса определяется временем обработки запроса самым медленным узлом. Попытаемся понять, как это влияет на производительность, и рассмотрим сервис, в котором 99-й процентиль задержки составляет 2 секунды. Это значит, что в среднем один из 100 запросов будет обработан с задержкой 2 секунды и более. Иными словами, обработка запроса займет 2 секунды и более с вероятностью 1 %. На первый взгляд, это совершенно приемлемо, так как только один из 100 запросов будет обрабатываться медленно. Но в систе-

мах, построенных по принципу Scatter/Gather, дела обстоят несколько иначе. Поскольку время обработки запроса определяется самым медленным ответом, нам приходится рассматривать не один запрос, а все запросы, распределенные между терминальными узлами системы.

Посмотрим, что произойдет, если распределить запросы на пять терминальных узлов. Вероятность того, что обработка запроса одним из терминальных узлов займет 2 секунды и более, равна 5 % (0,99 × 0,99 × 0,99 × 0,99 × 0,99 = 0,95). А это значит, что 99-й процентиль задержки единичного запроса становится 95-м процентилем задержки всей Scatter/Gather-системы. Дальше — хуже: если распределить нагрузку на 100 терминальных узлов, то *общая* средняя задержка обработки почти наверняка составит 2 секунды.

Из перечисленных выше проблем можно сделать несколько выводов:

- в силу накладных расходов, имеющих место в каждом узле, повышение степени параллелизма не всегда ускоряет работу системы;
- в силу «эффекта отстающего» повышение степени параллелизма не всегда ускоряет работу системы;
- 99-й процентиль производительности системы в Scatter/Gather-системах имеет существенно большее значение, нежели в других классах систем, так как один пользовательский запрос превращается во множество запросов к сервису.

«Эффект отстающего» влияет и на доступность системы. Если запрос распределяется на 100 терминальных узлов, а вероятность отказа любого из узлов равна 1 %, то почти наверняка ни один из запросов пользователей не будет успешно обработан.

Масштабирование Scatter/Gather-систем с учетом надежности и производительности

Как и в случае с другими шардированными системами, наличие единственной копии шардированной Scatter/Gather-системы — не лучшее архитектурное решение. Если в единственном экземпляре шарда происходит отказ, все Scatter/Gather-запросы будут отклоняться на время его недоступности, поскольку в рамках паттерна Scatter/Gather все запросы должны обрабатываться всеми терминальными узлами. Модернизация системы также сделает часть узлов временно недоступными, а значит, обновить систему при наличии запросов от пользователей тоже не получится. Наконец, вычислительная мощность системы в целом будет ограничена вычислительными возможностями каждого отдельного узла.

В конечном итоге все указанные факторы ограничивают масштабируемость вашей системы. Как уже было рассмотрено в предыдущих разделах, нельзя просто увеличить количество шардов, чтобы повысить вычислительную мощность системы, построенной на основе паттерна Scatter/Gather.

С учетом проблем с надежностью и масштабируемостью корректным будет реплицировать каждый отдельный шард. Таким образом, терминальные узлы будут реализованы не в виде отдельных шардов, а в виде реплицированных сервисов. Схема реплицированного шардированного паттерна Scatter/Gather изображена на рис. 7.4.

Нагрузка на терминальный узел системы равномерно распределяется между исправными репликами шарда. Это значит, что отказ реплик шарда не приведет к видимому отказу сервиса. Кроме того, в каждом реплицированном шарде можно

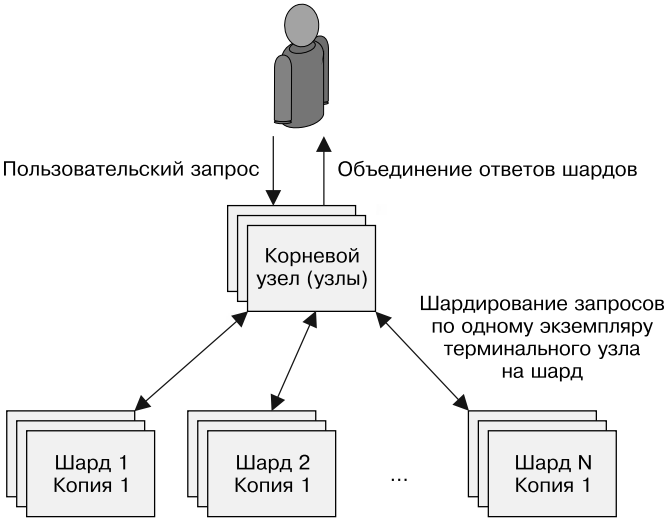

Рис. 7.4. Scatter/Gather-система с шардированием и репликацией

обновлять по одной реплике за раз, а значит, систему можно обновлять и под нагрузкой. В зависимости от того, насколько быстро нужно выполнять обновление, можно даже параллельно обновлять несколько реплик, находящихся в разных шардах.

8 Функции и событийно-ориентированная обработка

До настоящего момента мы рассматривали проектирование длительно работающих систем. Серверы, обрабатывающие пользовательские запросы, работают постоянно. Такой подход годится для высоконагруженных приложений, приложений, требующих большого объема памяти или фоновой обработки данных. Есть класс приложений, которые запускаются ненадолго — для обработки одного запроса или для того, чтобы отреагировать на конкретное событие. Такой запросно- или событийно-ориентированный подход к проектированию приложений начал в последнее время получать распространение по мере того, как крупные провайдеры стали предлагать продукты типа *«функция как сервис»* (function-as-a-service, FaaS). С недавних пор реализации FaaS начали работать на оркестраторах кластеров в частных облачных и физических средах. В этой

главе описываются новые архитектуры, возникающие в рамках данного подхода. В большинстве случаев FaaS представляет собой компонент более крупной архитектуры, а не самостоятельное решение.

Часто подход FaaS сопоставляют с бессерверными вычислениями. И хотя это так (в FaaS нет серверов), следует различать событийно-ориентированные FaaS и бессерверные вычисления в широком смысле. Действительно, бессерверные вычисления применимы к широкому спектру сервисов. К примеру, многопользовательские оркестраторы контейнеров («контейнер как сервис») являются бессерверными, но не являются событийно-ориентированными. И наоборот, FaaS-сервис с открытым исходным кодом, работающий на кластере физических компьютеров, принадлежащих вам и управляемых вами, является событийно-ориентированным, но не является бессерверным. Понимание этого различия позволяет вам определить, подходит ли для вашего приложения событийно-ориентированная либо бессерверная архитектура или обе сразу.

Как определить, когда полезен подход FaaS

Как и в случае с другими инструментами разработки распределенных систем, может возникнуть желание использовать частное решение (например, событийно-ориентированную обработку) в качестве универсального инструмента. Правда в том, что частное решение обычно решает частные задачи. В определенном контексте оно окажется мощным инструментом, но притягивание его за уши для решения общих проблем порождает сложные, хрупкие архитектуры. Поскольку подход FaaS еще нов, стоит уделить особое внимание рассмотрению его достоинств, недостатков, а также ситуаций, в которых его наиболее уместно применять.

Преимущества FaaS

Преимущества подхода FaaS в первую очередь касаются разработчиков. Он существенно сокращает расстояние между исходным текстом и запущенным сервисом. Поскольку, за исключением исходных текстов, нет необходимости создавать и разворачивать другие артефакты, FaaS упрощает переход от исходных текстов, открытых в текстовом редакторе на ноутбуке или в браузере, к запущенному в облаке приложению.

Масштабирование и управление развернутым кодом происходит автоматически. По мере возрастания трафика для его обработки создаются новые экземпляры функции. При отказе функции в результате отказа приложения или аппаратного обеспечения она автоматически перезапускается на другой машине.

Наконец, подобно контейнеру, функция — еще более мелкая составная часть распределенной системы. Функции не хранят состояние, и поэтому любая система, построенная на основе функций, по определению будет модульной и слабосвязанной, чем такая же система, собранная в один исполняемый файл. Но в этом свойстве также кроется сложность разработки распределенных систем на основе подхода FaaS. Слабая связность системы — одновременно ее преимущество и недостаток. В следующем разделе рассматриваются сложности и проблемы, сопутствующие разработке систем на основе подхода FaaS.

Проблемы разработки FaaS-систем

Как было сказано в предыдущем разделе, разработка систем с использованием подхода FaaS вынуждает делать компоненты системы слабосвязанными. Каждая функция независима по определению. Все взаимодействие осуществляется по сети. Экземпляры функций не имеют собственной памяти, следовательно, они требуют наличия общего хранилища для хранения

состояния. Принудительное ослабление связности элементов системы может повысить гибкость и скорость разработки сервисов, но в то же время может существенно осложнить ее поддержку.

В частности, довольно сложно увидеть исчерпывающую структуру сервиса, определить, как функции интегрируются друг с другом, понять, что и почему пошло не так в случае отказа. Кроме того, запросно-ориентированная и бессерверная природа функций означает, что некоторые проблемы будет сложно обнаружить. Рассмотрим в качестве примера следующие функции:

- `functionA()` вызывает `functionB()`;
- `functionB()` вызывает `functionC()`;
- `functionC()` вызывает `functionA()`.

Теперь рассмотрим, что происходит, когда в одну из этих функций поступает некоторый запрос. Начинается бесконечный цикл, который завершается только тогда, когда истекает время ожидания запроса (а может, и позже) или когда у вас заканчиваются деньги на оплату запросов к системе. Приведенный выше пример кажется надуманным, но такие ситуации довольно тяжело обнаружить в исходных текстах. Поскольку каждая функция сознательно отвязана от остальных, зависимость и взаимодействие функций явным образом не представлены в коде. Эти проблемы решаемы, и я надеюсь, что по мере созревания FaaS-технологий станут доступны инструменты анализа и отладки, дающие более полное понимание, как и почему приложение работает так, как оно работает.

При развертывании FaaS, по крайней мере пока, придется самостоятельно обеспечить строгий мониторинг и уведомление о состоянии приложений, чтобы вовремя обнаруживать и устранять нештатные ситуации, пока они не переросли в серьезные

проблемы. Сложность, привносимая мониторингом, несколько противоречит простоте развертывания FaaS-сервисов, но это тот барьер, который вашей команде придется преодолеть.

Потребность в фоновой обработке

FaaS по своей природе событийно-ориентированная модель построения приложений. Функции выполняются в ответ на дискретные события, их инициирующие.

Кроме того, в силу бессерверной реализации сервисов время выполнения экземпляра функции обычно ограничено. Это значит, что подход FaaS обычно не годится для приложений, требующих длительной фоновой обработки данных. В качестве примера можно привести такие задачи, как перекодирование видео, сжатие файлов журналов и другие долгосрочные вычисления с низким приоритетом. Во многих случаях есть возможность настроить искусственную генерацию событий по некоторому расписанию. Она хорошо подходит для реакции на временные события (например, чтобы разбудить кого-то текстовым сообщением), но предоставляемой ей инфраструктуры недостаточно для фоновых вычислений общего назначения. Чтобы учесть и их, необходимо запускать код в среде, поддерживающей постоянно работающие процессы. В общем случае это означает, что части приложения, которые осуществляют фоновую обработку, придется перевести с платы за количество запросов на плату за количество потребленных ресурсов.

Необходимость хранения данных в памяти

Кроме эксплуатационных проблем, у FaaS есть ряд архитектурных ограничений, делающих его малопригодным для некоторых классов приложений. Первое из них — необходимость загрузки значительного количества данных в память до обра-

ботки пользовательского запроса. Ряд служб (например, служба индексирования документов) требуют для своей работы загрузки в оперативную память большого количества данных. Даже при относительно высокой скорости хранилища загрузка большого количества данных может занять существенно больше времени, чем хотелось бы. Поскольку FaaS-функция может быть динамически инициализирована в ответ на пользовательский запрос, необходимость загрузки большого количества данных может существенно увеличить воспринимаемую пользователем задержку обработки его запроса. Как только вы создали FaaS-сервис, он может обрабатывать большое количество запросов, поэтому накладные расходы на загрузку амортизируются большим количеством обработанных запросов. Но если у вас запросов столько, что функция постоянно активна, вероятно, вы переплачиваете за количество обрабатываемых запросов.

Стоимость постоянного использования запросно-ориентированных вычислений

Ценовая модель облачных FaaS основана на плате за количество запросов. Такой подход хорош, если запросов к сервису немного — несколько штук в минуту или в час. В такой ситуации сервис преимущественно бездействует и с учетом платы за количество запросов вы платите только за то время, которое ваш сервис активно обрабатывает запросы. Напротив, если вы обслуживаете запросы с помощью постоянно работающего в контейнере или на виртуальной машине сервиса, то платите за процессорное время, которое тратится преимущественно на ожидание пользовательских запросов.

Однако по мере роста сервиса количество обрабатываемых запросов вырастает до такого уровня, что процессор все время занят их обработкой.

В этот момент плата за количество запросов начинает становиться невыгодной. Затраты на единицу процессорного времени облачных виртуальных машин сокращаются по мере добавления ядер, а также за счет резервирования ресурсов и скидок за длительное пользование. Затраты на оплату количества запросов обычно повышаются с ростом количества запросов.

Следовательно, по мере роста и эволюции вашего сервиса вы, вероятно, также станете по-другому использовать подход FaaS. Идеально будет масштабироваться FaaS-сервис с открытым кодом, работающий под управлением оркестратора контейнеров вроде Kubernetes. Таким образом, вы сможете сочетать преимущества FaaS с выгодной ценовой моделью виртуальных машин.

Паттерны FaaS

Для проектирования качественных систем жизненно важно понимать не только преимущества и недостатки FaaS-архитектур, но и то, как внедрить подход FaaS в распределенную систему. В данном разделе описываются несколько базовых паттернов внедрения FaaS.

Паттерн Decorator. Преобразование запроса или ответа

FaaS идеально подходит в том случае, когда нужны простые функции, которые обрабатывают входные данные, а затем передают их другим сервисам. Такого рода паттерн может использоваться для расширения или декорирования HTTP-запросов, передаваемых или принимаемых другим сервисом. Данный паттерн схематически изображен на рис. 8.1.

К слову, в языках программирования существует несколько аналогий данному паттерну. В частности, в Python есть *декораторы функций*, которые функционально похожи на декораторы за-

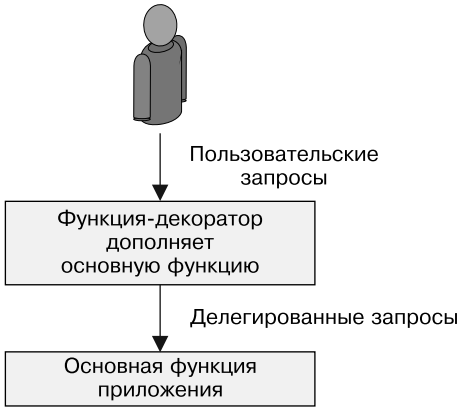

Рис. 8.1. Применение паттерна Decorator при проектировании HTTP API

просов или ответов. Поскольку декорирующие преобразования не хранят состояния и часто добавляются постфактум по мере развития сервиса, они идеально подходят для реализации в виде FaaS. Кроме того, легковесность FaaS означает, что можно экспериментировать с разными декораторами до тех пор, пока не найдется тот, который теснее интегрируется в реализацию сервиса.

Добавление значений по умолчанию во входные параметры HTTP RESTful API-запросов выгодно демонстрирует преимущества паттерна Decorator. Во многих API-запросах есть поля, которые необходимо заполнять разумными значениями, если они не были указаны вызывающей стороной. К примеру, вы хотите, чтобы по умолчанию поле хранило значение `true`. Этого трудно добиться с помощью классического JSON, поскольку в нем значение пустого поля по умолчанию равно `null`, что обычно интерпретируется как `false`. Чтобы решить эту проблему, можно добавить логику подстановки значений по умолчанию либо перед API-сервером, либо в коде самого приложения (например, `if (field == null) field = true`). Однако оба этих подхода неоптимальны, поскольку механизм подстановки значений по умолчанию концептуально

независим от обработки запроса. Вместо них мы можем использовать FaaS-паттерн Decorator, преобразующий запрос на пути между пользователем и реализацией сервиса.

Учитывая сказанное ранее в разделе об одноузловых паттернах, вам, возможно, стало интересно, почему мы не оформили сервис подстановки значений по умолчанию в виде контейнера-адаптера. Такой подход имеет смысл, но он также означает, что масштабирование сервиса подстановки значений по умолчанию и масштабирование самого API-сервиса становятся зависимы друг от друга. Подстановка значений по умолчанию — вычислительно легкая операция, и для нее, скорее всего, не понадобится много экземпляров сервиса.

В примерах данной главы мы будем использовать FaaS-фреймворк kubeless (https://github.com/kubeless/kubeless). Kubeless разворачивается поверх сервиса оркестратора контейнеров Kubernetes. Если вы уже подготовили Kubernetes-кластер, то приступайте к установке Kubeless, который можно загрузить с соответствующего сайта (https://github.com/kubeless/kubeless/releases). Как только у вас появился исполняемый файл kubeless, установить его в кластер можно командой kubeless install.

Kubeless устанавливается как сторонняя API-надстройка Kubernetes. А это значит, что после установки его можно будет использовать в рамках инструмента командной строки kubectl. К примеру, развернутые в кластере функции можно будет увидеть, выполнив команду kubectl get functions. На данный момент в вашем кластере не развернуто ни одной функции.

Практикум. Подстановка значений по умолчанию до обработки запроса

Продемонстрировать полезность паттерна Decorator в FaaS можно на примере подстановки значений по умолчанию в RESTful-

вызов для параметров, значения которых не были заданы пользователем. С помощью FaaS это делается довольно просто. Функция подстановки значений по умолчанию написана на языке Python:

```
# Простая функция-обработчик, подставляющая значения
# по умолчанию
def handler(context):
  # Получаем входное значение
  obj = context.json
  # Если поле "name" отсутствует, инициализировать его
  # случайной строкой
  if obj.get("name", None) is None:
    obj["name"] = random_name()
  # Если отсутствует поле 'color', установить его
  # значение в 'blue'
  if obj.get("color", None) is None:
    obj["color"] = "blue"
  # Выполнить API-вызов с учетом значений параметров
  # по умолчанию
  # и вернуть результат
  return call_my_api(obj)
```

Сохраните эту функцию в файл под названием `defaults.py`. Не забудьте заменить вызов `call_my_api` вызовом нужного вам API. Эту функцию подстановки значений по умолчанию можно зарегистрировать в качестве kubeless-функции следующей командой:

```
kubeless function deploy add-defaults \
    --runtime python27 \
    --handler defaults.handler \
    --from-file defaults.py \
    --trigger-http
```

Чтобы ее протестировать, можно использовать инструмент `kubeless`:

```
kubeless function call add-defaults --data '{"name": "foo"}'
```

Паттерн Decorator показывает, насколько просто адаптировать и расширять существующие API дополнительными возможностями вроде валидации или подстановки значений по умолчанию.

Обработка событий

Большинство систем являются запросно-ориентированными — они обрабатывают непрерывные потоки пользовательских и API-запросов. Несмотря на это, существует довольно много событийно-ориентированных систем. Различие между запросом и событием, как мне кажется, кроется в понятии *сессии*. Запросы представляют собой части более крупного процесса взаимодействия (сессии). В общем случае каждый пользовательский запрос есть часть процесса взаимодействия с веб-приложением либо API в целом. *События* видятся мне более «одноразовыми», асинхронными по своей природе. События важны и должны соответствующим образом обрабатываться, но они оказываются вырваны из основного контекста взаимодействия и ответ на них приходит лишь спустя некоторое время. Примером события может служить подписка пользователя на некоторый сервис, что вызовет отправку приветственного письма; загрузка файла в общую папку, что приведет к отправке уведомлений всем пользователям данной папки; или даже подготовка компьютера к перезагрузке, что приведет к уведомлению оператора или автоматизированной системы о том, что необходимо предпринять соответствующие действия.

Поскольку эти события в значительной степени независимы и не имеют внутреннего состояния, а частота их весьма изменчива, они идеально подходят для работы в событийно-ориентированных FaaS-архитектурах. Их часто разворачивают рядом с «боевым» сервером приложений для обеспечения дополнительных возможностей или для фоновой обработки данных в ответ на возникающие события. Кроме того, поскольку к сервису постоянно добавляются новые типы обрабатываемых событий, простота развертывания функций делает их подходящими для реализации обработчиков событий. А так как каждое событие концептуально независимо от остальных, вынужденное

ослабление связей внутри системы, построенной на основе функций, позволяет *снизить* ее концептуальную сложность, позволяя разработчику сосредоточиться на шагах, необходимых для обработки только одного конкретного типа событий.

Конкретный пример интеграции событийно-ориентированного компонента к существующему сервису — реализация двухфакторной аутентификации. В данном случае событием будет вход пользователя в систему. Сервис может генерировать для этого действия событие и передавать его функции-обработчику. Обработчик на основе переданного кода и контактных данных пользователя отправит ему аутентификационный код в виде текстового сообщения.

Практикум. Реализация двухфакторной аутентификации

Двухфакторная аутентификация указывает, что для входа в систему пользователю надо что-то, что он знает (например, пароль), и что-то, что он имеет (например, номер телефона). Двухфакторная аутентификация намного лучше просто пароля, поскольку злоумышленнику для получения доступа придется украсть и ваш пароль, и номер вашего телефона.

При планировании реализации двухфакторной аутентификации нужно обработать запрос на генерацию случайного кода, зарегистрировать его в службе входа в систему и отправить сообщение пользователю. Можно добавить код, реализующий эту функциональность, непосредственно в саму службу входа в систему. Это усложняет систему, делает ее более монолитной. Отправка сообщения должна выполняться одновременно с кодом, генерирующим веб-страницу входа в систему, что может привнести определенную задержку. Эта задержка ухудшает качество взаимодействия пользователя с системой.

Лучше будет создать FaaS-сервис, который бы асинхронно генерировал случайное число, регистрировал его в службе входа в систему и отправлял на телефон пользователя. Таким образом, сервер входа в систему может просто выполнить асинхронный запрос к FaaS-сервису, который параллельно выполнит относительно медленную задачу регистрации и отправки кода.

Для того чтобы увидеть, как это работает, рассмотрим следующий код:

```
def two_factor(context):
  # Сгенерировать случайный шестизначный код
  code = random.randint(100000, 999999)

  # Зарегистрировать код в службе входа в систему
  user = context.json["user"]
  register_code_with_login_service(user, code)

  # Для отправки сообщения воспользуемся библиотекой Twillio
  account = "my-account-sid"
  token = "my-token"
  client = twilio.rest.Client(account, token)

  user_number = context.json["phoneNumber"]
  msg = "Здравствуйте, {}, ваш код аутентификации:
        {}.".format(user, code)
  message = client.api.account.messages.create(to=user_number,
                                  from_="+12065251212",
                                  body=msg)
  return {"status": "ok"}
```

Затем зарегистрируем FaaS в kubeless:

```
kubeless function deploy add-two-factor \
    --runtime python27 \
    --handler two_factor.two_factor \
    --from-file two_factor.py \
    --trigger-http
```

Экземпляр этой функции может асинхронно порождаться из клиентского кода на JavaScript после ввода пользователем пра-

вильного пароля. Веб-интерфейс может немедленно отобразить страницу для ввода кода, а пользователь, как только получит код, может сообщить его службе входа в систему, в которой этот код уже зарегистрирован.

Итак, подход FaaS существенно облегчил разработку простого, асинхронного, событийно-ориентированного сервиса, который инициируется при входе пользователя в систему.

Событийные конвейеры

Существует ряд приложений, которые, по сути, проще рассматривать как конвейер слабо связанных событий. Конвейеры событий часто напоминают старые добрые блок-схемы. Их можно представить в виде ориентированного графа синхронизации связанных событий. В рамках паттерна Event Pipeline узлы соответствуют функциям, а дуги, их соединяющие, — HTTP-запросам или другого рода сетевым вызовам.

Между элементами контейнера, как правило, нет общего состояния, но может быть общий контекст или другая точка отсчета, на основе которой будет выполняться поиск в хранилище.

Какова же разница между таким конвейером и микросервисной архитектурой? Есть два важных различия. Первое и самое главное различие между сервисами-функциями и постоянно работающими сервисами состоит в том, что событийные конвейеры, по сути, управляются событиями. Микросервисная архитектура же, напротив, подразумевает набор постоянно работающих сервисов. Кроме того, событийные конвейеры могут быть асинхронными и связывать разнообразные события. Сложно представить, как можно интегрировать одобрение заявки в системе Jira в микросервисное приложение. В то же время нетрудно представить, как оно интегрируется в событийный конвейер.

В качестве примера рассмотрим конвейер, в котором исходным событием будет загрузка кода в систему контроля версий. Это событие вызывает пересборку кода. Сборка может занять несколько минут, после чего создается событие, инициирующее функцию тестирования собранного приложения. В зависимости от успешности сборки функция тестирования предпринимает разные действия. Если сборка прошла успешно, создается заявка, которая должна быть одобрена человеком, чтобы новая версия приложения вошла в эксплуатацию. Закрытие заявки служит сигналом к вводу новой версии в эксплуатацию. Если сборка завершилась неудачно, в Jira делается заявка об обнаруженной ошибке, а конвейер завершает работу.

Практикум. Реализация конвейера для регистрации нового пользователя

Рассмотрим задачу реализации последовательности действий для регистрации нового пользователя. При создании новой учетной записи всегда выполняется целый ряд действий, например отправка приветственного электронного письма. Есть также ряд действий, которые могут выполняться не каждый раз, например подписка на e-mail-рассылку о новых версиях продукта (также известную как спам).

Один из подходов подразумевает создание монолитного сервиса *создания новых учетных записей*. При таком подходе одна команда разработчиков несет ответственность за весь сервис, который к тому же развертывается как единое целое. Это затрудняет проведение экспериментов и внесение изменений в процесс взаимодействия пользователя с приложением.

Рассмотрим реализацию входа пользователя в систему как событийный конвейер из нескольких FaaS-сервисов. При таком разделении функция создания пользователя понятия не имеет,

что происходит во время входа пользователя в систему. У нее есть два списка:

- список необходимых действий (например, отправка приветственного электронного письма);
- список необязательных действий (например, подписка на рассылку).

Каждое из этих действий также реализуется в виде FaaS, а список действий есть не что иное, как список HTTP-функций обратного вызова. Стало быть, функция создания пользователя имеет следующий вид:

```
def create_user(context):
  # Безусловный вызов всех необходимых обработчиков
  for key, value in required.items():
    call_function(value.webhook, context.json)

  # Необязательные обработчики выполняются
  # при соблюдении определенных условий
  for key, value in optional.items():
  if context.json.get(key, None) is not None:
    call_function(value.webhook, context.json)
```

Каждый из обработчиков теперь также можно реализовать по принципу FaaS:

```
def email_user(context):
  # Получить имя пользователя
  user = context.json['username']

  msg = 'Здравствуйте, {}, спасибо, что воспользовались нашим
        замечательным сервисом!".format(user)

  send_email(msg, contex.json['email])

def subscribe_user(context):
  # Получить имя пользователя
  email = context.json['email']
  subscribe_user(email)
```

Декомпозированный таким образом FaaS-сервис становится значительно проще, содержит меньше строк кода и сосредоточен на реализации одной конкретной функции. Микросервисный подход упрощает написание кода, но может привести к сложностям при развертывании и управлении тремя разными микросервисами. Здесь подход FaaS проявляет себя во всей красе, поскольку в результате его использования становится очень просто управлять небольшими фрагментами кода. Визуализация процесса создания пользователя в виде событийного конвейера позволяет также в общих чертах понять, что именно происходит во время входа пользователя в систему, просто проследив изменение контекста от функции к функции в рамках конвейера.

9 Выбор владельца

Ранее рассмотренные паттерны касались преимущественно распределения запросов с целью увеличения количества обрабатываемых в секунду запросов, сокращения времени обработки запроса, необходимости передачи состояния. Последняя глава в разделе о многоузловых паттернах проектирования касается масштабирования закрепления ресурсов. Во многих системах фигурирует такое понятие, как *владение*, — когда конкретный процесс владеет конкретной задачей. Такое мы уже видели при рассмотрении систем с фиксированным и «горячим» шардированием. Конкретные экземпляры шардированного сервиса владеют соответствующими частями пространства ключей шардирования.

В контексте единственного сервера владения добиться несложно, поскольку только одно приложение устанавливает владение и делает это с помощью хорошо известного механизма блокировок, что обеспечивает единоличное владение актора конкретным шардом или контекстом. Ограничение владения одним

приложением снижает масштабируемость, так как репликация становится невозможна. Снижается и надежность, поскольку при отказе приложения оно становится на некоторый период времени недоступно. Следовательно, когда в системе требуется механизм владения, необходимо разработать распределенную систему установления владения.

Общая схема распределенного владения приводится на рис. 9.1. На схеме показаны три экземпляра, каждый из которых может быть владельцем, или хозяином. Сначала владельцем будет первый экземпляр. Если в нем случится ошибка, владельцем станет третий экземпляр.

Наконец, первый экземпляр восстанавливает работу и снова входит в группу, но при этом третий экземпляр все так же остается владельцем.

Распределенное владение — одновременно самая сложная и самая важная часть проектирования надежной распределенной системы.

Как определить, нужен ли выбор владельца

Простейшая форма владения — наличие единственной копии сервиса. Поскольку в каждый момент времени работает лишь один экземпляр сервиса, он по умолчанию является владельцем всего и никакой выбор не нужен. Преимущество такого подхода — в простоте создания и развертывания приложения. Его недостатки заключаются в снижении доступности и надежности такого сервиса. Однако для многих приложений простота данного подхода перевешивает потерю надежности. Давайте рассмотрим этот момент подробнее.

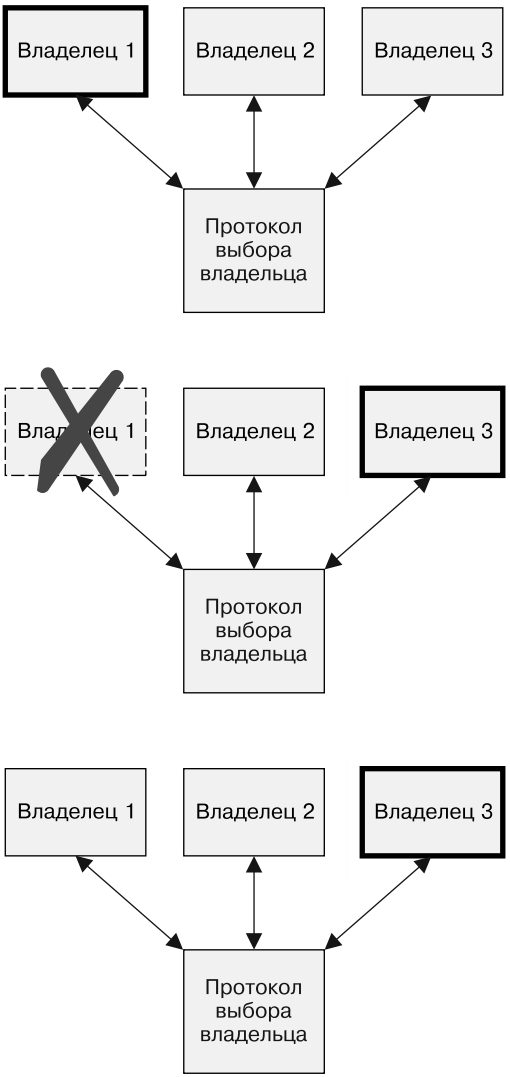

Рис. 9.1. Протокол выбора владельца в действии: сначала владельцем является первый экземпляр сервиса, а в случае его отказа полномочия берет на себя третий экземпляр

Допустим, вы запускаете сервис-одиночку в оркестраторе контейнеров вроде Kubernetes. В таком случае у вас есть следующие гарантии.

- Если контейнер откажет, он будет перезапущен.
- Если контейнер зависнет, а у вас реализована проверка работоспособности, он также будет перезапущен.
- Если произойдет аппаратный сбой, контейнер будет перемещен на другую машину.

Благодаря наличию этих гарантий сервис-одиночка, работающий под управлением оркестратора контейнеров, как правило, имеет достаточно высокий процент времени безотказной работы. Чтобы уточнить, насколько высок «достаточно высокий» процент, рассмотрим, что происходит в случае каждого из перечисленных отказов. Если запущенный в контейнере процесс откажет, приложение будет перезапущено через несколько секунд. Предположим, контейнер сбоит раз в день. Это примерно соответствует доступности на уровне 3–4 девяток (99,9–99,99 %) — 2 секунды недоступности в день примерно равны 99,99 % доступности. Если контейнер отказывает реже, то доступность будет еще лучше. При отказе оборудования Kubernetes потребуется некоторое время, чтобы понять, что машина вышла из строя, после чего он переместит контейнер на другую машину. Допустим, это займет 5 минут. Если каждая машина в кластере отказывает раз в день, то доступность сервиса составляет две девятки, или 99 %. Честно сказать, если у вас в кластере каждая машина отказывает раз в день, то низкая доступность сервиса — наименьшая из ваших проблем.

Конечно, стоит учесть, что недоступность сервиса может быть вызвана и другими причинами. При развертывании ПО на загрузку и установку новой версии приложения требуется не-

которое время. Старая и новая версии синглтона не могут работать одновременно, поэтому на время обновления придется отключить старую версию приложения, а при большом размере образа процесс может занять несколько минут. Следовательно, при ежедневном развертывании, на которое требуется 2 минуты, доступность сервиса будет на уровне двух девяток (99 %), а при ежечасном — ниже 90 %. Развертывание, конечно, можно ускорить путем предварительной загрузки образа на обновляемую машину. Это уменьшит время развертывания очередной версии до нескольких секунд, но привнесет дополнительную сложность, которой мы избегали изначально.

Несмотря на это, существует ряд приложений (например, фоновая асинхронная обработка), в которых такой компромисс между уровнем доступности и простотой приложения допустим. Один из ключевых аспектов проектирования распределенной системы — следить, чтобы распределенность не слишком ее усложняла. Но есть и ситуации, когда высокая доступность (четыре девятки и больше) является критической характеристикой приложения. В таких системах необходимо иметь несколько экземпляров сервиса, среди которых только один является владельцем. Проектирование такого рода систем описывается в последующих разделах.

Основы процесса выбора владельца

Представим себе сервис Foo в трех экземплярах: Foo-1, Foo-2 и Foo-3. Допустим, есть также некоторый объект Bar, которым одновременно может владеть только один экземпляр сервиса. Этот экземпляр часто называется *владельцем*. Соответственно, процесс, описывающий то, как выбрать начального владельца и в случае отказа текущего — очередного владельца, называется *выбором владельца*.

Есть два способа реализовать выбор владельца. Первый — создать распределенный алгоритм согласования вроде Paxos или RAFT. Сложность этих алгоритмов выводит их за рамки книги и делает их дальнейшее рассмотрение нецелесообразным. Реализация какого-нибудь из них сравнима с реализацией блокировок с помощью ассемблерных инструкций сравнения и обмена. Из нее выйдет хорошая задачка для университетского курса информатики, но на практике так делать не стоит.

К счастью, есть ряд распределенных хранилищ типа «ключ — значение», в которых эти алгоритмы уже реализованы. В общем и целом эти системы предоставляют реплицированные, надежные хранилища, а также примитивы, необходимые для построения сложных механизмов выбора и блокировки. К таким хранилищам относятся, к примеру, etcd, ZooKeeper и consul. К базовым примитивам, предоставляемым подобными системами, относится операция сравнения с заменой для конкретного ключа. Если вы ранее не сталкивались с операцией сравнения с заменой, то знайте: она представляет собой атомарную операцию следующего вида:

```
var lock = sync.Mutex{}
var store = map[string]string{}

func compareAndSwap(key, nextValue, currentValue string)
  (bool, error) {
  lock.Lock()
  defer lock.Unlock()
  if _, found := store[key]; !found {
    if len(currentValue) == 0 {
      store[key] = nextValue
      return true, nil
    }
    return false, fmt.Errorf("Для ключа %s ожидалось значение %s,
      но было обнаружено пустое значение", key, currentValue)
  }
```

```
  if store[key] == currentValue {
    store[key] = nextValue
    return true, nil
  }
  return false, nil
}
```

Операция сравнения с заменой атомарно записывает новое значение ключа, если его текущее значение совпадает с ожидаемым. В противном случае она возвращает `false`. Если значения не существует, а `currentValue` не равно `null`, она возвращает ошибку.

Вдобавок к операции сравнения с заменой хранилища типа «ключ — значение» позволяют устанавливать для ключа срок действия (TTL, time-to-live). Как только он истекает, значение ключа обнуляется.

Этих двух функций достаточно, чтобы реализовать множество примитивов синхронизации.

Практикум. Развертывание etcd

Etcd (https://coreos.com/etcd/docs/latest/) — распределенный сервис блокировок, разработанный в рамках проекта CoreOS. Он довольно устойчив и хорошо себя зарекомендовал в различных проектах, в том числе в Kubernetes.

Развертывание etcd существенно упростилось благодаря двум проектам с открытым исходным кодом:

- helm (https://helm.sh/) — менеджеру пакетов Kubernetes, поддерживаемому Microsoft Azure;
- оператору etcd (https://coreos.com/blog/introducing-the-etcd-operator.html), разработанному в рамках проекта CoreOS.

Операторы — интересное направление, исследуемое в рамках проекта CoreOS. Оператор — это программа, работающая в рамках оркестратора контейнеров с единственной целью — управление работой одного или нескольких приложений. Оператор отвечает за создание, масштабирование и поддержку функционирования приложения. Пользователи настраивают приложение, задавая его желаемое состояние с помощью API. К примеру, оператор etcd отвечает за сервис etcd. Операторы — относительно новая задумка, но они представляют собой важное направление в разработке надежных распределенных систем.

Чтобы развернуть оператор etcd в CoreOS, воспользуемся менеджером пакетов helm. Helm — менеджер пакетов, являющийся частью Kubernetes и разработанный Deis. Microsoft Azure поглотила Deis в 2017 году, и Microsoft с тех пор продолжает поддерживать open-source-разработку данного проекта.

Если вы впервые используете helm, вам понадобится установить его согласно инструкции по адресу https://github.com/kubernetes/helm/releases.

После запуска helm в своей среде вы можете установить оператор etcd следующим образом:

```
# Инициализация helm
helm init

# Установка оператора etcd
helm install stable/etcd-operator
```

После установки оператор etcd создает собственный класс Kubernetes-ресурсов, представляющих собой etcd-кластеры. Оператор запущен, но кластеры etcd еще не созданы. Для создания etcd-кластера необходимо составить следующую декларативную конфигурацию:

```
apiVersion: "etcd.coreos.com/v1beta1"
kind: "Cluster"
```

```
metadata:
  # Имя может быть любым
  name: "my-etcd-cluster"
spec:
  # size может принимать значения 1, 3 и 5
  size: 3
  # Устанавливаемая версия etcd
  version: "3.1.0"
```

Сохраните данную конфигурацию в файл `etcd-cluster.yaml` и создайте кластер с помощью такой команды:

```
kubectl create -f etcd-cluster.yaml
```

После создания кластера оператор etcd автоматически создаст pod-группы с копиями кластера. Копии кластера можно увидеть, выполнив команду:

```
kubectl get pods
```

После запуска всех трех копий адреса точек доступа к ним можно получить с помощью такой команды:

```
export ETCD_ENDPOINTS=kubectl get endpoints example-etcd-cluster
"-o=jsonpath={.subsets[*].addresses[*].ip}:2379,"
```

Теперь в кластер можно что-нибудь записать следующей командой:

```
kubectl exec my-etcd-cluster-0000 -- sh -c "ETCD_API=3 etcdctl
--endpoints=${ETCD_ENDPOINTS} set foo bar"
```

Реализация блокировок

Простейшая форма синхронизации — взаимоисключающая блокировка, также известная как *мьютекс* (mutual exclusion, mutex). С блокировками знаком каждый, кто хоть раз занимался созданием параллельных программ для одной машины. Для распределенных систем работают те же принципы. Распределенные блокировки используют не память и ассемблерные

инструкции, а рассмотренные ранее функции распределенных хранилищ типа «ключ — значение».

Как и с блокировками памяти, первый шаг — установление блокировки:

```
func (Lock l) simpleLock() boolean {
  // Сравнение с заменой "1" на "0"
  locked, _ = compareAndSwap(l.lockName, "1", "0")
  return locked
}
```

Блокировки может еще не существовать, поскольку мы первые, кто ее запрашивает. Учтем и это:

```
func (Lock l) simpleLock() boolean {
  // Сравнение с заменой "1" на "0"
  locked, error = compareAndSwap(l.lockName, "1", "0")
  // Блокировки не существует, пытаемся записать "1"
  // поверх несуществующего значения
  if error != nil {
    locked, _ = compareAndSwap(l.lockName, "1", nil)
  }
  return locked
}
```

Традиционные реализации блокировки останавливают исполнение на время действия блокировки, поэтому нам еще понадобится что-то вроде:

```
func (Lock l) lock() {
  while (!l.simpleLock()) {
    sleep(2)
  }
}
```

Проблема данной реализации, несмотря на ее простоту, состоит в том, что после снятия блокировки придется ждать минимум 1 секунду, чтобы установить ее снова. К счастью, многие хранилища «ключ — значение» позволяют следить за изменениями,

не прибегая к опросу сервера, поэтому реализация может выглядеть таким образом:

```
func (Lock l) lock() {
  while (!l.simpleLock()) {
    waitForChanges(l.lockName)
  }
}
```

Реализация снятия блокировки будет выглядеть так:

```
func (Lock l) unlock() {
  compareAndSwap(l.lockName, "0", "1")
}
```

Может показаться, что работа сделана, но помните, что речь идет о распределенной системе. Установивший блокировку процесс может отказать до ее снятия, а значит, снять ее будет уже некому. В такой ситуации система зависнет. Для разрешения данной проблемы воспользуемся возможностью назначить ключу срок действия (TTL). Изменим функцию `simpleLock` так, чтобы она всегда записывала TTL. Таким образом, если в течение данного времени процесс не снимет блокировку, она будет снята автоматически.

```
func (Lock l) simpleLock() boolean {
  // Сравнение с заменой "1" на "0"
  locked, error = compareAndSwap(l.lockName, "1", "0", l.ttl)
  // Блокировки не существует, пытаемся записать "1"
  // поверх несуществующего значения
  if error != nil {
    locked, _ = compareAndSwap(l.lockName, "1", nil, l.ttl)
  }
  return locked
}
```

При использовании распределенных блокировок чрезвычайно важно, чтобы выполняемая вами обработка данных укладывалась в срок действия блокировки. Хорошим приемом будет запуск сторожевого таймера при установке блокировки.

Сторожевой таймер аварийно прекратит исполнение программы, если вы не сняли блокировку в период TTL.

Добавив к блокировкам TTL, мы привнесли ошибку в функцию разблокировки. Рассмотрим следующую ситуацию.

1. Процесс 1 устанавливает блокировку с TTL, равным *t*.
2. В силу некоторых причин процесс работает очень медленно и не справляется с задачей за время *t*.
3. Срок действия блокировки истекает.
4. Процесс 2 ставит блокировку, поскольку процесс 1 ее потерял в связи с истечением TTL.
5. Процесс 1 заканчивает работу и вызывает функцию *разблокировки*.
6. Процесс 3 ставит блокировку.

На данный момент процесс 1 считает, что снял установленную им блокировку. Он не знает, что потерял блокировку из-за истечения TTL, и фактически снял блокировку, установленную процессом 2. Тут появляется процесс 3 и ставит блокировку. Теперь процессы 2 и 3 оба считают, что поставили блокировку, и тут-то и начинается потеха.

К счастью, хранилища «ключ — значение» поддерживают *версионность ресурсов*. Версия ресурса меняется при каждой записи. Функция блокировки может записывать версию ресурса и расширять операцию `compareAndSwap` проверкой на совпадение не только значения, но и версии ресурса. Функция блокировки будет выглядеть следующим образом:

```
func (Lock l) simpleLock() boolean {
  // Сравнение с заменой "1" на "0"
  locked, l.version, error = compareAndSwap(l.lockName, "1",
                                            "0", l.ttl)
```

```
  // Блокировки не существует, пытаемся записать "1"
  // поверх несуществующего значения
  if error != null {
    locked, l.version, _ = compareAndSwap(l.lockName, "1",
                                          null, l.ttl)
  }
  return locked
}
```

Функция разблокировки будет выглядеть так:

```
func (Lock l) unlock() {
  compareAndSwap(l.lockName, "0", "1", l.version)
}
```

Таким образом, блокировка снимается, только если TTL не истек.

Практикум. Реализация блокировок в etcd

При реализации блокировок в etcd ключи можно применять для имен блокировок, а начальные условия записи задавать таким образом, чтобы в каждый момент времени существовал только один владелец блокировки. Для простоты поработаем с утилитой командной строки etcdctl для установки и снятия блокировки. В действительности же вам следует воспользоваться языком программирования. Клиенты etcd есть для всех популярных языков программирования.

Начнем с создания блокировки `my-lock`:

```
kubectl exec my-etcd-cluster-0000 -- sh -c \
  "ETCD_API=3 etcdctl --endpoints=${ETCD_ENDPOINTS} set
   my-lock unlocked"
```

Это создаст в etcd блокировку `my-lock` с начальным значением `unlocked`.

Допустим, Алиса и Боб хотят установить блокировку `my-lock`. Они пытаются записать свои имена в блокировку, исходя из того, что она изначально снята.

Алиса выполняет команду:

```
kubectl exec my-etcd-cluster-0000 -- sh -c \
  "ETCD_API=3 etcdctl --endpoints=${ETCD_ENDPOINTS} \
    set --swap-with-value unlocked my-lock alice"
```

тем самым ставя блокировку. Теперь блокировку пытается установить Боб:

```
kubectl exec my-etcd-cluster-0000 -- sh -c \
  "ETCD_API=3 etcdctl --endpoints=${ETCD_ENDPOINTS} \
    set --swap-with-value unlocked my-lock bob"
Error: 101: Compare failed ([unlocked != alice]) [6]
```

Как видим, попытка Боба установить блокировку оказалась неудачной, так как блокировка была поставлена Алисой.

Чтобы снять блокировку, Алиса записывает значение `unlocked` вместо `alice`:

```
kubectl exec my-etcd-cluster-0000 -- sh -c \
  "ETCD_API=3 etcdctl --endpoints=${ETCD_ENDPOINTS} \
    set --swap-with-value alice my-lock unlocked"
```

Реализация владения

Блокировки хороши для установления временного владения некоторым важным компонентом. Иногда может понадобиться установить владение ресурсом на все время работы компонента. Возьмем, к примеру, кластер Kubernetes с высокой доступностью. Допустим, в нем есть несколько экземпляров планировщика, но только один из них активно принимает решения. Кроме того, как только процесс становится активным планировщиком, он остается таковым до момента отказа.

Одним из подходов будет поднятие TTL до очень большой величины — скажем, недели или даже больше. Существенный недостаток такого подхода состоит в том, что в случае отказа текущего владельца блокировки новый будет выбран только по истечении TTL, неделю спустя.

Вместо этого мы можем создать *возобновляемую блокировку*, которая будет периодически возобновляться владельцем, позволяя процессу держать блокировку в течение произвольного периода времени.

Расширим предыдущую версию функции блокировки возможностью возобновления ее владельцем:

```
func (Lock l) renew() boolean {
  locked, _ = compareAndSwap(l.lockName, "1", "1", l.version, ttl)
  return locked
}
```

Для того чтобы поддерживать блокировку в течение неопределенного промежутка времени, придется выполнять эти действия в отдельном потоке. Обратите внимание, что блокировка возобновляется каждые TTL / 2 секунды, чтобы снизить риск ее случайного истечения в силу особенностей подсчета времени:

```
for {
  if !l.renew() {
    handleLockLost()
  }
  sleep(ttl/2)
}
```

Сперва, конечно, необходимо реализовать функцию `handleLockLost()`, которая прекращает деятельность, требовавшую блокировки. В рамках оркестратора контейнеров проще всего будет завершить приложение — пусть оркестратор его перезапустит. Это безопасно, поскольку другой экземпляр сервиса

в то же время перехватит блокировку. Когда перезапущенное приложение возобновит работу, оно станет ожидать освобождения блокировки.

Практикум. Реализация аренды в etcd

Вернемся к ранее рассмотренному примеру работы с блокировками, в частности к флагу `--ttl=`, используемому при создании и возобновлении блокировки. Флаг `--ttl=` определяет интервал времени, спустя который блокировка удаляется. Поскольку по истечении TTL секунд блокировка исчезает, будем исходить не из того, что начальное значение блокировки `unlocked`, а из того, что отсутствие блокировки означает, что ресурс свободен. Воспользуемся командой `mk` вместо `set`. `etcdctl mk` отработает успешно только в том случае, если ключа в данный момент *не* существует.

Следовательно, чтобы установить арендованную блокировку, Алиса выполняет такую команду:

```
kubectl exec my-etcd-cluster-0000 -- \
  sh -c "ETCD_API=3 etcdctl --endpoints=${ETCD_ENDPOINTS} \
    --ttl=10 mk my-lock alice"
```

Таким образом создается арендованная блокировка продолжительностью 10 секунд.

Чтобы возобновить блокировку, надо выполнить следующую команду:

```
kubectl exec my-etcd-cluster-0000 -- \
  sh -c "ETCD_API=3 etcdctl --endpoints=${ETCD_ENDPOINTS} \
    set --ttl=10 --swap-with-value alice my-lock alice"
```

Может показаться странным, что Алиса постоянно записывает свое имя в блокировку, но именно так она сможет продлить ее на очередные 10 секунд.

Если TTL по какой-то причине истечет, возобновления блокировки не произойдет и Алисе придется вновь ее устанавливать. В это время Боб тоже может установить блокировку. Для поддержки владения Бобу также потребуется устанавливать и возобновлять блокировку каждые 10 секунд.

Параллельный доступ к данным

Даже при использовании описанных ранее механизмов блокировки остается возможность того, что в течение небольшого промежутка времени два экземпляра сервиса будут думать, что они оба установили блокировку. Чтобы понять, как это может произойти, представьте, что текущий владелец блокировки становится настолько перегруженным, что перестает работать по нескольку минут подряд. Такое может случиться на машинах, где одновременно исполняется слишком много задач. В этом случае блокировка истечет и ее перехватит другая копия сервиса. Процессор освобождает копию сервиса, которая была заблокирована предыдущим владельцем. Очевидно, вскоре будет вызвана функция `handleLockLost()`, но в течение небольшого периода времени копия будет считать, что все еще владеет блокировкой.

Хотя вероятность такого события невелика, системы необходимо создавать с учетом указанных обстоятельств. Сперва убедимся, что блокировка все еще активна:

```
func (Lock l) isLocked() boolean {
  return l.locked && l.lockTime + 0.75 * l.ttl > now()
}
```

Если данная функция выполняется раньше любого кода, требующего блокировок, вероятность одновременного существования двух владельцев значительно снижается, но, что важно, не исчезает полностью. Блокировка может истечь между проверкой наличия блокировки и выполнением защищенного кода. Чтобы

защититься от таких случаев, система, к которой обращается копия сервиса, должна проверить, что отправивший запрос сервис действительно является владельцем. Для этого в хранилище, кроме состояния блокировки, должно фиксироваться имя хоста копии — владельца блокировки. Так другие участники процесса смогут проверить, что копия, называющая себя владельцем, на самом деле им является.

Схема системы показана на рис. 9.2. Блокировкой владеет `shard2`, и, когда рабочему узлу отправляется запрос, тот уточняет на сервере блокировок, действительно ли `shard2` является текущим владельцем.

Во втором случае `shard2` утратил владение блокировкой, но еще не осознал этого и поэтому продолжает отправлять запросы рабочему узлу. На сей раз, получив запрос от `shard2`, рабочий узел уточняет его статус на сервере блокировок. Узнав, что он больше не является владельцем, рабочий узел отвергает этот и последующие его запросы.

Еще одна сложность состоит в том, что владение может быть получено, потеряно и затем получено заново. В таком случае запрос выполнится успешно, хотя должен быть отвергнут. Чтобы понять, как такое возможно, рассмотрим последовательность событий.

1. Шард 1 становится владельцем.
2. Шард 1 отправляет в качестве мастера запрос в момент времени `T1`.
3. Сеть зависает, и доставка `R1` задерживается.
4. Шард 1 превышает время действия блокировки, и ее перехватывает шард 2.
5. Шард 2 становится владельцем и отправляет запрос `R2` в момент времени `T2`.

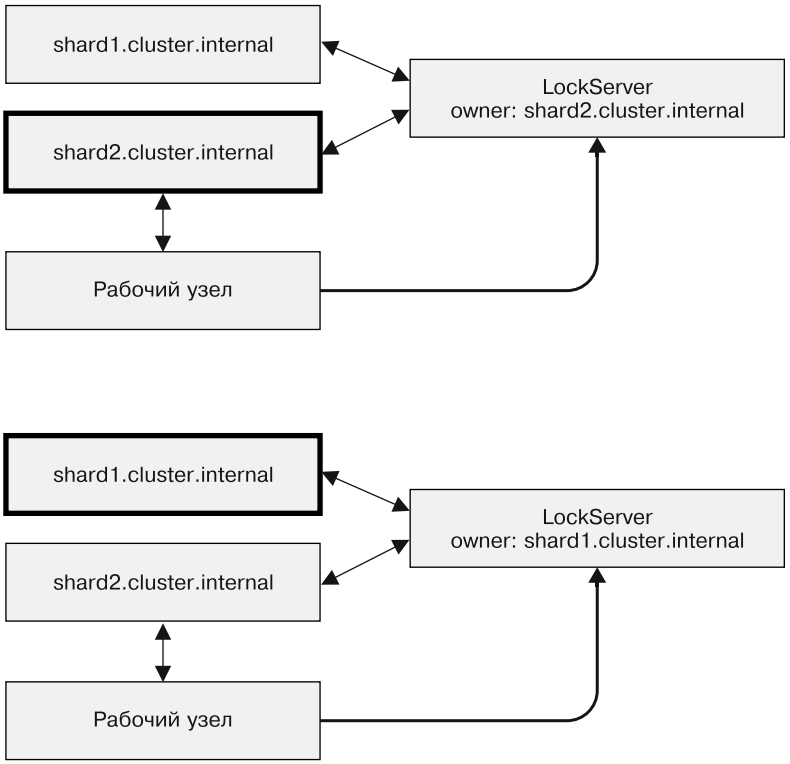

Рис. 9.2. Рабочий узел проверяет, что отправитель запроса действительно является текущим владельцем шарда

6. Запрос `R2` получен и обработан.
7. Шард 2 отказывает, и владение возвращается к шарду 1.
8. Запрос `R1` наконец достигает цели. Шард 1 в данный момент является владельцем, поэтому его запрос принимается к исполнению.

Эта последовательность событий кажется коварной, но в реальной большой системе такие вещи случаются с пугающей частотой. К счастью, ситуация похожа на ранее рассмотренную

проблему, которая была решена с использованием версионирования ресурсов в etcd. Здесь можно поступить так же. Вдобавок к фиксации текущего владельца можно с каждым запросом отправлять версию ресурса. `R1` из предыдущего примера становится равен `(R1, Version1)`. Теперь при получении запроса уточняется не только текущий владелец, но и версия ресурса. Запрос отклоняется при любом несовпадении. Так мы «подлатали» данный пример.

Часть III
Паттерны проектирования систем пакетных вычислений

В предыдущей части рассматривались паттерны проектирования надежных, постоянно работающих приложений. В этой части описываются паттерны проектирования систем пакетной обработки. В отличие от постоянно работающих приложений пакетные процессы обычно работают в течение небольшого промежутка времени. Примерами процессов пакетной обработки могут служить следующие процессы: генерация сводки по данным пользовательской телеметрии, анализ данных продаж для ежедневной или еженедельной отчетности.

Пакетные процессы характеризуются быстрой обработкой большого объема данных с максимально возможным применением параллелизма. Самый известный паттерн проектирования распределенных систем — MapReduce — уже успел стать отдельным самостоятельным направлением. Есть, однако, и другие паттерны, полезные для пакетной обработки. Они рассматриваются в последующих главах.

10 Системы на основе очередей задач

Простейшая форма пакетной обработки — *очередь задач*. В системе с очередью задач есть набор задач, которые должны быть выполнены. Каждая задача полностью независима от остальных и может быть обработана без всяких взаимодействий с ними. В общем случае цель системы с очередью задач — обеспечить выполнение каждого этапа работы в течение заданного промежутка времени. Количество рабочих потоков увеличивается либо уменьшается сообразно изменению нагрузки. Схема обобщенной очереди задач представлена на рис. 10.1.

Система на основе обобщенной очереди задач

Очередь задач — идеальный пример, демонстрирующий всю мощь паттернов проектирования распределенных систем. Большая часть логики работы очереди задач никак не зависит от

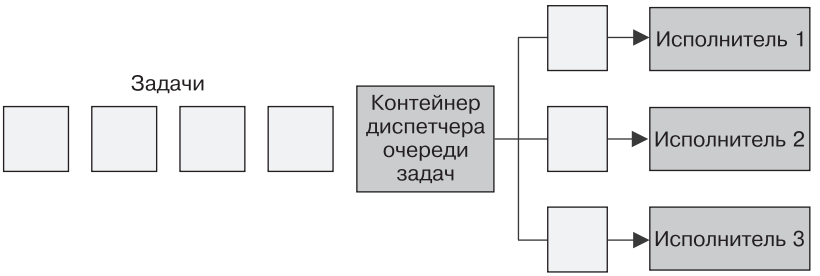

Рис. 10.1. Обобщенная очередь задач

рода выполняемой работы. Во многих случаях то же касается и доставки самих задач. Проиллюстрируем данное утверждение с помощью очереди задач, изображенной на рис. 10.1. Посмотрев на нее еще раз, определите, какие ее функции могут быть предоставлены совместно используемым *набором контейнеров*. Становится очевидным, что большая часть реализации контейнеризованной очереди задач может использоваться широким спектром пользователей.

Построение очереди задач на основе контейнеров требует согласования интерфейсов между библиотечными контейнерами и контейнерами с пользовательской логикой. В рамках контейнеризованной очереди задач выделяется два интерфейса: интерфейс контейнера-источника, предоставляющего поток задач, требующих обработки, и интерфейс контейнера-исполнителя, который знает, как их обрабатывать.

Интерфейс контейнера-источника

Любая очередь задач функционирует на основе набора задач, требующих обработки. В зависимости от конкретного приложения, реализованного на базе очереди задач, существует множество источников задач, в нее попадающих. Но после получения набора задач схема работы очереди оказывается

довольно простой. Следовательно, мы можем отделить специфичную для приложения логику работы источника задач от обобщенной схемы обработки очереди задач. Вспомнив ранее рассмотренные паттерны групп контейнеров, здесь можно разглядеть реализацию паттерна Ambassador. Контейнер обобщенной очереди задач является главным контейнером приложения, а специфичный для приложения контейнер-источник является послом, транслирующим запросы контейнера-диспетчера очереди конкретным исполнителям задач. Данная группа контейнеров изображена на рис. 10.2.

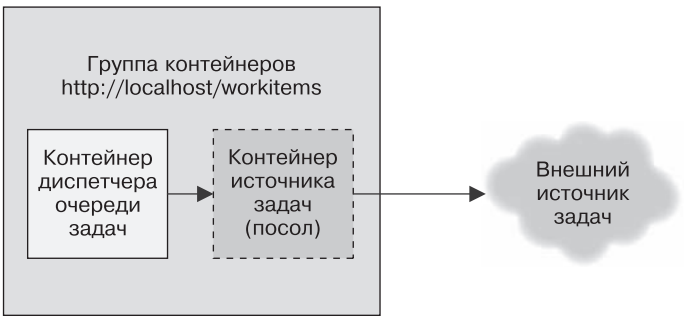

Рис. 10.2. Группа контейнеров, реализующая очередь задач

К слову, хотя контейнер-посол специфичен для приложения (что очевидно), существует также ряд обобщенных реализаций API источника задач. Например, источником может служить список фотографий, находящихся в некотором облачном хранилище, набор файлов на сетевом диске или даже очередь в системах, работающих по принципу «публикация/подписка», таких как Kafka или Redis. Несмотря на то что пользователи могут выбирать наиболее подходящие под свою задачу контейнеры-послы, им следует использовать обобщенную «библиотечную» реализацию самого контейнера. Так будет минимизирован объем работы и максимизировано повторное использование кода.

API очереди задач. Учитывая механизм взаимодействия очереди задач и зависящего от приложения контейнера-посла, нам следует сформулировать формальное определение интерфейса между двумя контейнерами. Существует много разных протоколов, но HTTP RESTful API просты в реализации и являются стандартом де-факто для подобных интерфейсов. Очередь задач ожидает, что в контейнере-после будут реализованы следующие URL:

- GET http://localhost/api/v1/items;
- GET http://localhost/api/v1/items/<item-name>.

Зачем добавлять v1 в определение API, спросите вы? Появится ли когда-нибудь вторая версия интерфейса? Выглядит нелогично, но расходы на версионирование API при первоначальном его определении минимальны. Проводить же соответствующий рефакторинг позже станет крайне дорого. Возьмите за правило добавлять версии ко всем API, даже если не уверены, что они когда-либо изменятся. Береженого Бог бережет.

URL `/items/` возвращает список всех задач:

```
{
  kind: ItemList,
  apiVersion: v1,
  items: [
    "item-1",
    "item-2",
    ….
  ]
}
```

URL `/items/<item-name>` предоставляет подробную информацию о конкретной задаче:

```
{
  kind: Item,
```

```
  apiVersion: v1,
  data: {
    "some": "json",
    "object": "here",
  }
}
```

Обратите внимание, что в API не предусмотрено никаких механизмов фиксации факта выполнения задачи. Можно было бы разработать более сложный API и переложить большую часть реализации на контейнер-посол. Помните, однако, что наша цель — сосредоточить как можно большую часть общей реализации внутри диспетчера очереди задач. В этой связи диспетчер очереди задач должен сам следить за тем, какие задачи уже обработаны, а какие еще предстоит обработать.

Из этого API мы получаем сведения о конкретной задаче, а затем передаем значение поля `item.data` интерфейса контейнера-исполнителя.

Интерфейс контейнера-исполнителя

Как только диспетчер очереди получил очередную задачу, он должен поручить ее некоторому исполнителю. Это второй интерфейс в обобщенной очереди задач. Сам контейнер и его интерфейс немного отличаются от интерфейса контейнера-источника по нескольким причинам. Во-первых, это «одноразовый» API. Работа исполнителя начинается с единственного вызова, и в течение жизненного цикла контейнера больше никаких вызовов не выполняется. Во-вторых, контейнер-исполнитель и диспетчер очереди задач находятся в разных группах контейнеров. Контейнер-исполнитель запускается посредством API оркестратора контейнеров в своей собственной группе. Это значит, что диспетчер очереди задач должен выполнить удаленный вызов, чтобы инициировать работу контейнера-исполнителя. Это также значит, что придется быть более осторожными

в вопросах безопасности, так как злонамеренный пользователь кластера может загрузить его лишней работой.

В контейнере-источнике для отправки списка задач диспетчеру задач мы использовали простой HTTP-вызов. Так было сделано исходя из того, что данный API-вызов нужно совершать несколько раз, а вопросы безопасности не учитывались, поскольку все работало в рамках localhost. API контейнера-исполнителя необходимо вызывать лишь однажды и важно убедиться, что другие пользователи системы не могут добавить работы исполнителям хоть случайно, хоть по злому умыслу. Следовательно, для контейнера-исполнителя будем использовать файловый API. При создании мы передадим контейнеру переменную среды под названием `WORK_ITEM_FILE`, значение которой ссылается на файл во внутренней файловой системе контейнера. Этот файл содержит данные о задаче, которую необходимо выполнить. Такого рода API, как показано ниже, может быть реализован Kubernetes-объектом `ConfigMap`. Его можно смонтировать в группу контейнеров как файл (рис. 10.3).

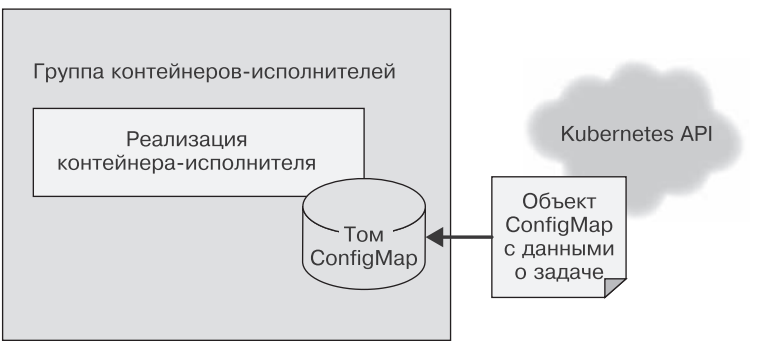

Рис. 10.3. API контейнера-исполнителя

Такой механизм файлового API проще реализовать с помощью контейнера. Исполнитель в рамках очереди задач часто

представляет собой простой сценарий командной оболочки, обращающийся к нескольким инструментам. Нецелесообразно поднимать для управления задачами целый веб-сервер — это приводит к усложнению архитектуры. Как и в случае с источниками задач, большая часть контейнеров-исполнителей будет представлять собой специализированные контейнеры для определенных задач, но есть и обобщенные исполнители, применимые для решения нескольких разных задач.

Рассмотрим пример контейнера-исполнителя, который скачивает файл из облачного хранилища, выполняет над ним сценарий командной оболочки, а затем копирует результат обратно в облачное хранилище. Такой контейнер может быть по большей части общим, но в качестве параметра ему может передаваться конкретный сценарий. Таким образом, большая часть кода работы с файлом может быть повторно использована многими пользователями/очередями задач. Конечному пользователю необходимо лишь предоставить сценарий, содержащий специфику обработки файла.

Общая инфраструктура очередей задач

Что остается внедрить в повторно используемой реализации очереди, если реализации двух ранее описанных интерфейсов контейнеров у вас уже есть? Базовый алгоритм работы очереди задач довольно прост.

1. Загрузить из контейнера-источника доступные на данный момент задачи.
2. Уточнить состояние очереди задач на предмет того, какие задачи уже выполнены или еще выполняются.
3. Для каждой из нерешенных задач породить контейнеры-исполнители с соответствующим интерфейсом.
4. При успешном завершении контейнера-исполнителя зафиксировать, что задача выполнена.

Этот алгоритм прост на словах, но в действительности его не так легко реализовать. К счастью, оркестратор Kubernetes имеет несколько возможностей, которые значительно упрощают его реализацию. А именно: в Kubernetes есть объект `Job`, который позволяет обеспечить надежную работу очереди задач. Объект `Job` можно настроить так, чтобы он запускал соответствующий контейнер-исполнитель либо разово, либо пока задача не будет успешно выполнена. Если контейнер-исполнитель настроить, чтобы он выполнялся до завершения задачи, то, даже когда машина в кластере откажет, задача в конце концов будет выполнена успешно.

Таким образом, построение очереди задач существенно упрощается, так как оркестратор берет на себя ответственность за надежное исполнение задач.

Кроме того, Kubernetes позволяет аннотировать задачи, что дает нам возможность пометить каждый объект-задачу названием обрабатываемого элемента очереди задач. Становится проще различать задачи, обрабатываемые и завершенные как успешно, так и с ошибкой.

Это значит, что мы можем реализовать очередь задач поверх оркестратора Kubernetes, не используя собственного хранилища. Все это существенно упрощает задачу построения инфраструктуры очереди задач.

Следовательно, подробный алгоритм работы контейнера — диспетчера очереди задач выглядит следующим образом.

Повторять бесконечно.

1. Получить список задач посредством интерфейса контейнера — источника задач.
2. Получить список заданий, обслуживающих данную очередь задач.

3. Выделить на основе этих списков перечень необработанных задач.
4. Для каждой необработанной задачи создать объект `Job`, который порождает соответствующий контейнер-исполнитель.

Приведу сценарий на языке Python, реализующий такую очередь:

```
import requests
import json
from kubernetes import client, config
import time

namespace = "default"

def make_container(item, obj):
    container = client.V1Container()
    container.image = "my/worker-image"
    container.name = "worker"
    return container

def make_job(item):
    response =
      requests.get("http://localhost:8000/items/{}".format(item))
    obj = json.loads(response.text)
    job = client.V1Job()
    job.metadata = client.V1ObjectMeta()
    job.metadata.name = item
    job.spec = client.V1JobSpec()
    job.spec.template = client.V1PodTemplate()
    job.spec.template.spec = client.V1PodTemplateSpec()
    job.spec.template.spec.restart_policy = "Never"
    job.spec.template.spec.containers = [
        make_container(item, obj)
    ]
    return job

def update_queue(batch):
    response = requests.get("http://localhost:8000/items")
```

```
    obj = json.loads(response.text)
    items = obj['items']

    ret = batch.list_namespaced_job(namespace, watch=False)

    for item in items:
        found = False
        for i in ret.items:
            if i.metadata.name == item:
                found = True
        if not found:
            # Функция создает объект Job, пропущена
            # для краткости
            job = make_job(item)
            batch.create_namespaced_job(namespace, job)

config.load_kube_config()
batch = client.BatchV1Api()

while True:
    update_queue(batch)
    time.sleep(10)
```

Практикум. Реализация генератора миниатюр видеофайлов

В качестве примера использования очереди задач рассмотрим задачу генерации миниатюр видеофайлов. На основе этих миниатюр пользователи принимают решение о том, какие видео они хотят посмотреть.

Для реализации миниатюр понадобится два контейнера. Первый — для источника задач. Проще всего будет размещать задачи на общем сетевом диске, подключенном, например, по NFS (Network File System, сетевая файловая система). Источник задач получает список файлов в этом каталоге и передает их вызывающей стороне.

Приведу простую программу на NodeJS:

```
const http = require('http');
const fs = require('fs');

const port = 8080;
const path = process.env.MEDIA_PATH;

const requestHandler = (request, response) => {
    console.log(request.url);
    fs.readdir(path + '/*.mp4', (err, items) => {
        var msg = {
            'kind': 'ItemList',
            'apiVersion': 'v1',
            'items': []
        };
        if (!items) {
              return msg;
        }
        for (var i = 0; i < items.length; i++) {
            msg.items.push(items[i]);
        }
        response.end(JSON.stringify(msg));
    });
}

const server = http.createServer(requestHandler);

server.listen(port, (err) => {
    if (err) {
        return console.log('Ошибка запуска сервера', err);
    }

    console.log(`сервер запущен на порте ${port}`)
});
```

Данный источник определяет список фильмов, подлежащих обработке. Для извлечения миниатюр используется утилита ffmpeg.

Можно создать контейнер, запускающий такую команду:

```
ffmpeg -i ${INPUT_FILE} -frames:v 100 thumb.png
```

Команда извлекает один из каждых 100 кадров (параметр `-frames:v 100`) и сохраняет его в формате PNG (например, `thumb1.png`, `thumb2.png` и т. д.).

Подобного рода обработку можно реализовать на основе существующего Docker-образа `ffmpeg`. Популярностью пользуется образ `jrottenberg/ffmpeg` (https://hub.docker.com/r/jrottenberg/ffmpeg/).

Определив простой контейнер-источник и еще более простой контейнер-исполнитель, нетрудно заметить, какие преимущества имеет обобщенная, контейнерно-ориентированная система управления очередью. Она существенно сокращает время между проектированием и реализацией очереди задач.

Динамическое масштабирование исполнителей

Рассмотренная ранее очередь задач хорошо подходит для обработки заданий по мере их поступления, но может привести к скачкообразной нагрузке на ресурсы кластера оркестратора контейнеров. Это хорошо, когда у вас много разных видов заданий, создающих нагрузочные пики в разное время и тем самым равномерно распределяющих нагрузку на кластер во времени.

Но если у вас недостаточно видов нагрузки, подход «то густо, то пусто» к масштабированию очереди задач может потребовать резервирования дополнительных ресурсов для поддержки всплесков нагрузки. В остальное время ресурсы будут простаивать, без надобности опустошая ваш кошелек.

Для решения данной проблемы можно ограничить общее количество объектов `Job`, порождаемых очередью задач. Это естественным образом ограничит количество параллельно обрабатываемых заданий и, следовательно, снизит использование ресурсов при пиковой нагрузке. С другой стороны, увеличится длительность исполнения каждой отдельной задачи при высокой нагрузке на кластер.

Если нагрузка носит скачкообразный характер, это нестрашно, поскольку для выполнения скопившихся задач можно пользоваться интервалами простоя. Однако если устойчивая нагрузка слишком высока, очередь задач не будет успевать обрабатывать приходящие задания и на их выполнение будет затрачиваться все больше и больше времени.

В такой ситуации вам придется динамически подстраивать максимальное количество параллельно выполняемых задач и, соответственно, доступных вычислительных ресурсов для поддержания необходимого уровня производительности. К счастью, есть математические формулы, позволяющие определить, когда необходимо масштабировать очередь задач для обработки большего количества запросов.

Рассмотрим очередь задач, в которой новое задание появляется в среднем раз в минуту, а его выполнение занимает в среднем 30 секунд. Такая очередь в состоянии справиться с потоком приходящих в нее заданий. Даже если одномоментно придет большой пакет заданий, образовав затор, то со временем затор будет ликвидирован, поскольку до поступления очередного задания очередь успевает обработать в среднем два задания.

Если же новое задание приходит каждую минуту и на обработку одного задания уходит в среднем 1 минута, то такая система

идеально сбалансирована, но при этом плохо реагирует на изменения в нагрузке. Она в состоянии справиться с всплесками нагрузки, но на это ей потребуется довольно много времени. У системы не будет простоя, но не будет и резерва машинного времени, чтобы скомпенсировать долгосрочное повышение скорости поступления новых задач. Для поддержания стабильности системы необходимо иметь резерв на случай долгосрочного роста нагрузки или непредвиденных задержек при обработке заданий.

Наконец, рассмотрим систему, в которой поступает одно задание в минуту, а обработка задания занимает две минуты. Такая система будет постоянно терять производительность. Длина очереди задач будет расти вместе с задержкой между поступлением и обработкой задач (и степенью раздраженности пользователей).

За значениями этих двух показателей необходимо постоянно следить. Усреднив время между поступлением заданий за длительный период времени, например на основе количества заданий за сутки, получим оценку *межзадачного интервала*. Необходимо также следить за средней продолжительностью обработки задания (без учета времени, проведенного им в очереди). В стабильной очереди задач среднее время обработки задачи должно быть меньше межзадачного интервала. Чтобы обеспечить выполнение такого условия, необходимо динамически подстраивать количество доступных очереди вычислительных ресурсов. Если задания обрабатываются параллельно, то время обработки следует разделить на количество параллельно обрабатываемых заданий. К примеру, если одно задание обрабатывается минуту, но параллельно обрабатываются четыре задачи, то эффективное время обработки одной задачи составляет 15 секунд, а значит, межзадачный интервал должен составлять не менее 16 секунд.

Такой подход позволяет без труда создать модуль масштабирования очереди задач в сторону увеличения. Масштабирование в сторону уменьшения несколько проблематичнее. Тем не менее можно использовать те же расчеты, что и ранее, дополнительно заложив определяемый эвристическим путем резерв вычислительных ресурсов. К примеру, можно снижать количество параллельно выполняемых задач до тех пор, пока время обработки одного задания не составит 90 % от межзадачного интервала.

Паттерн Multi-Worker

Одна из основных тем данной книги — использование контейнеров с целью инкапсуляции и повторного применения кода. Она актуальна и для паттернов построения очередей задач, описываемых в данной главе. Помимо контейнеров, управляющих самой очередью, повторно использовать можно и группы контейнеров, образующих реализацию исполнителей. Допустим, каждое задание в очереди вам необходимо обработать тремя разными способами. Например, обнаружить на фотографии лица, сопоставить их с конкретными людьми, а затем размыть соответствующие части изображения. Можно поместить всю обработку в один контейнер-исполнитель, но это одноразовое решение, которое невозможно будет использовать повторно. Для замазывания на фото чего-нибудь еще, например машин, придется с нуля создавать контейнер-исполнитель.

Возможности такого рода повторного использования можно добиться путем применения *паттерна Multi-Worker*, который фактически является частным случаем паттерна Adapter, описанного в начале книги. Паттерн Multi-Worker преобразует набор контейнеров в один общий контейнер с программным интерфейсом контейнера-исполнителя. Этот общий контейнер делегирует обработку нескольким отдельным, повторно

используемым контейнерам. Данный процесс схематически изображен на рис. 10.4.

Рис. 10.4. Агрегирующий паттерн Multi-Worker, реализованный в виде группы контейнеров

Благодаря повторному использованию кода путем комбинирования контейнеров-исполнителей снижаются трудозатраты людей, проектирующих распределенные системы пакетной обработки.

11 Событийно-ориентированная пакетная обработка

В предыдущей главе мы рассмотрели общую инфраструктуру для организации очередей задач, а также несколько простых примеров приложений, их использующих. Очереди задач хорошо подходят для однократного преобразования одного набора входных данных в один набор выходных данных.

Однако существует ряд приложений пакетной обработки, в которых может понадобиться выполнить несколько преобразований либо породить из одного набора входных данных несколько наборов выходных данных в разных форматах. В таких случаях приходится связывать очереди задач так, что выходные данные одной очереди задач становятся входными данными для другой очереди (или даже нескольких очередей) и т. д. За счет этого образуется последовательность шагов обработки, в которой каждая последующая очередь задач реагирует на событие завершения обработки в очереди предыдущего шага.

Такого рода событийно-ориентированные системы часто называются *системами с потоком задач*, поскольку они основаны на потоке задач в направленном ациклическом графе взаимосвязанных этапов обработки данных. Такая система схематически изображена на рис. 11.1.

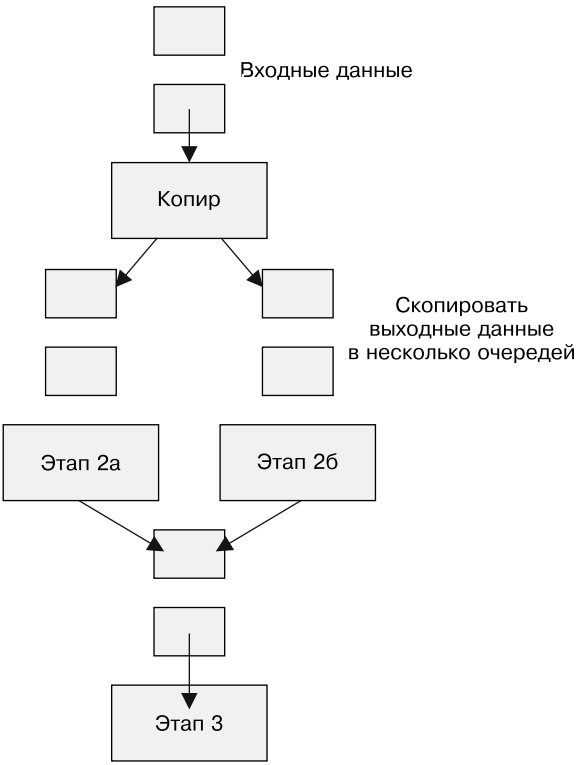

Рис. 11.1. В данном потоке задачи копируются в несколько параллельно обрабатываемых очередей (этапы 2а, 2б), а затем снова объединяются в общую очередь (этап 3)

Простейшее приложение такого рода систем подразумевает передачу выходных данных одной очереди на вход другой очереди. По мере усложнения систем появляются различные паттерны

связи очередей задач. Для понимания работы системы в целом чрезвычайно важно разбираться в этих паттернах. Принцип работы событийно-ориентированной очереди задач похож на принцип работы событийно-ориентированного FaaS-сервиса. Без общей схемы взаимодействия очередей друг с другом будет трудно в полной мере понять, как работает система.

Паттерны событийно-ориентированной обработки

Помимо простейшей очереди задач, описанной в предыдущей главе, существует ряд паттернов связывания очередей задач. Простейший из них, когда выходные данные одной очереди становятся входными данными другой очереди, достаточно прост и не требует рассмотрения. Рассмотрим паттерны, которые требуют согласованной работы нескольких очередей задач либо модификации выходных данных одной из очередей.

Паттерн Copier

Первый паттерн координации очередей задач — Copier. Задача контейнера-копира — преобразовать исходный поток задач в несколько идентичных параллельных потоков. Этот паттерн полезен, когда над входными данными задачи необходимо выполнить несколько различных видов действий. Возьмем, к примеру, рендеринг видео. Рендеринг может осуществляться во множество различных форматов, в зависимости от того, где видео будет демонстрироваться. Для воспроизведения с жесткого диска может использоваться разрешение 4K, для потоковой трансляции по сети — 1080p, а для пользователей с медленным мобильным Интернетом — еще более низкое разрешение. Можно также сгенерировать анимированную миниатюру в формате GIF для использования в интерфейсе плеера. Под каждый

формат рендеринга будет отведена отдельная очередь, но входные данные для любого исполнителя будут идентичны.

Применение паттерна Copier для перекодирования видео схематически изображено на рис. 11.2.

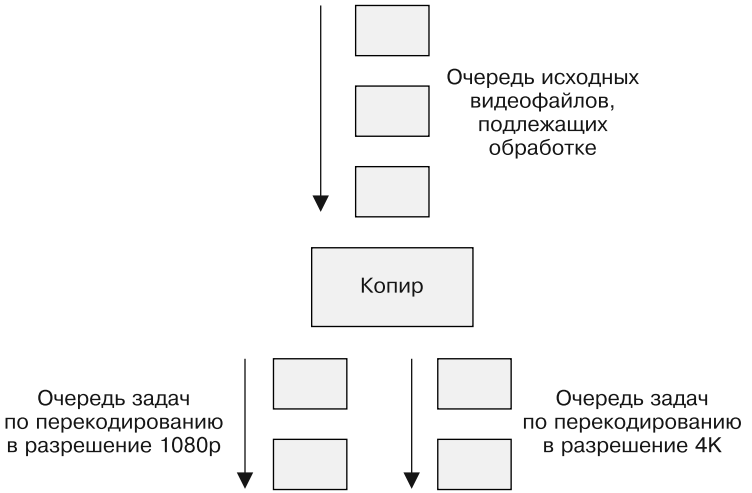

Рис. 11.2. Применение паттерна Copier для перекодирования видео

Паттерн Filter

Второй паттерн событийно-ориентированной пакетной обработки называется Filter. Его цель — уменьшить поток задач за счет фильтрации задач, не отвечающих определенным критериям. В качестве примера рассмотрим создание пакетного потока задач, обрабатывающего регистрацию нового пользователя. Некоторые пользователи при регистрации ставят флажок, означающий их согласие на получение по электронной почте рекламных рассылок и другой информации. В таком потоке задач фильтр должен пропускать только тех пользователей, которые явно подписались на получение рассылки.

В идеале контейнер-фильтр следует реализовывать в виде контейнера-посла, обертывающего существующий источник заданий. Исходный источник заданий предоставляет полный список заданий, подлежащих обработке, а контейнер-фильтр сокращает этот список на основе условия фильтрации и выдает только те элементы, которые ему удовлетворяют. Пример такого использования паттерна Adapter схематически приведен на рис. 11.3.

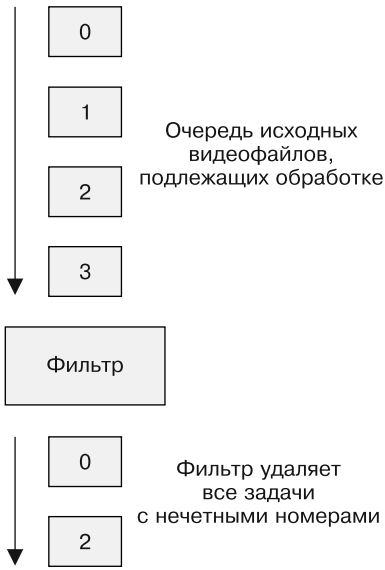

Рис. 11.3. Пример реализации паттерна Filter, когда удаляются все нечетные задания

Паттерн Splitter

Иногда может понадобиться не просто отфильтровать и отбросить ненужные задачи, а разделить их на два подмножества, каждое из которых попадет в свою очередь. Для такой задачи понадобится контейнер-делитель. Задача делителя, как

и фильтра, — вычислить значение некоторого критерия. Задачи, не удовлетворяющие данному критерию, в отличие от фильтра, не отсеиваются, а помещаются в другую очередь.

Примером приложения паттерна Splitter может служить обработка онлайн-заказов, уведомление о доставке которых пользователь получает либо по электронной почте, либо текстовым сообщением. Элементами очереди задач в данном случае станут подлежащие доставке заказы. Контейнер-делитель будет определять каждый заказ в одну из двух очередей — на отправку уведомления по электронной почте либо текстовым сообщением. Контейнер-делитель может также играть роль копира, отправляющего задачи в несколько очередей. Такое может произойти, когда при оформлении заказа пользователь выбрал оба типа уведомлений. Интересно также отметить, что делитель может быть реализован в виде комбинации копира и двух фильтров. Делитель, однако же, позволяет решить данную задачу более компактным образом.

Пример использования паттерна Splitter для рассылки уведомлений о доставке приведен на рис. 11.4.

Паттерн Sharder

Паттерн Sharder — несколько более общая форма паттерна Splitter. Задача шардера потока задач, во многом как и шардирующего сервера, рассмотренного несколькими главами ранее, — разделить очередь задач на несколько равных частей с помощью некоторой шардирующей функции. Необходимость в шардировании потока задач может возникнуть по нескольким причинам. Одна из основных — повышение надежности. При шардировании очереди задач отказ одного из потоков задач из-за неудачного обновления, отказа инфраструктуры или по любой другой причине повлияет лишь на часть пользователей вашего сервиса.

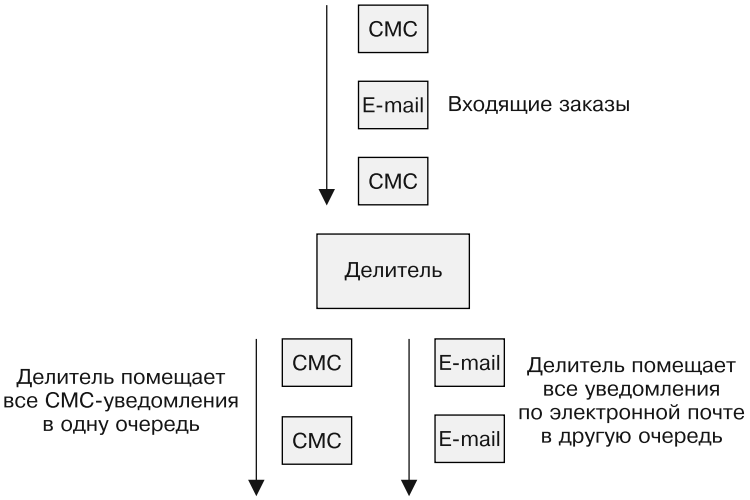

Рис. 11.4. Пример реализации паттерна пакетной обработки Splitter для рассылки уведомлений о доставке посредством двух очередей задач

Представьте, к примеру, что вы неудачно обновили контейнер-исполнитель, в результате чего все его экземпляры отказали и очередь задач перестала обрабатываться. Если задачи обрабатываются только одной очередью, то сервис окажется полностью недоступным для всех пользователей. Если бы вы шардировали очередь на четыре части, то у вас была бы возможность организовать поэтапное развертывание контейнера-исполнителя. Предположим, вы обнаруживаете отказ на первом этапе развертывания. При шардировании на четыре части отказ повлияет только на четверть пользователей вашего сервиса.

Еще один довод в пользу шардирования — более равномерное распределение нагрузки на вычислительные ресурсы. Если вам не особенно важно, за обработку каких задач будет отвечать конкретный центр обработки данных (ЦОД), шардером можно воспользоваться для распределения задач между несколькими ЦОД, чтобы выровнять нагрузку серверов в них. Что касается

обновлений, распределение очереди задач между несколькими точками отказа повышает надежность, позволяя избежать отказа всех серверов в конкретном ЦОД или целом регионе.

Шардированная очередь, работающая в штатном режиме, показана на рис. 11.5.

Рис. 11.5. Пример реализации шардированной очереди, работающей в штатном режиме

Если в силу отказов количество рабочих шардов уменьшилось, алгоритм шардирования динамически перестраивается на распределение работы только между рабочими шардами, даже если осталась только одна очередь задач. Это показано на рис. 11.6.

Паттерн Merger

Последним среди паттернов событийно-ориентированных систем пакетной обработки рассмотрим паттерн Merger. Он выполняет

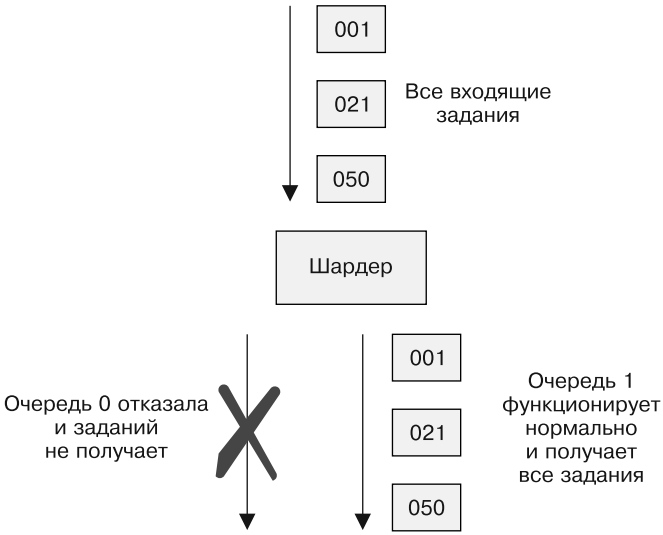

Рис. 11.6. При отказе одной из очередей задач оставшиеся переходят в другую очередь

задачу, обратную задаче паттерна Copier, — объединяет две очереди в одну общую. Допустим, в вашем проекте много разных репозиториев исходного кода, коммиты в которые происходят одновременно. Вы хотите выполнить тестирование и сборку каждого из них. Создавать отдельную инфраструктуру для сборки репозитория — плохо масштабируемое решение. Каждый из репозиториев можно смоделировать в виде очереди задач, служащей источником задач-коммитов. Все эти источники задач можно объединить в один интегрированный источник с помощью адаптера-объединителя. Такой объединенный поток коммитов служит единственным источником задач для системы сборки, выступающей исполнителем. Контейнер-объединитель является частным случаем реализации паттерна Adapter. Такой адаптер преобразует потоки задач от нескольких контейнеров-источников в общий поток задач. Схема паттерна Multi-Adapter приведена на рис. 11.7.

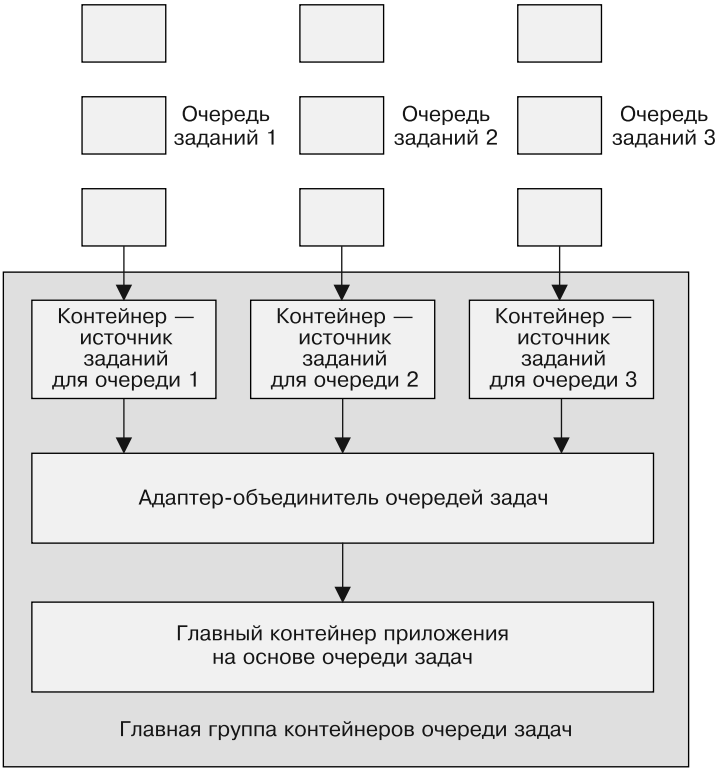

Рис. 11.7. Использование нескольких уровней контейнеров для объединения нескольких очередей задач в одну общую

Практикум. Создание событийно-ориентированного потока задач для регистрации нового пользователя

Пример конкретного потока задач позволяет показать, как эти паттерны можно объединить для получения полноценной рабочей системы. В данном примере рассматривается задача регистрации нового пользователя.

Представьте, что наша «воронка» получения пользователей работает в два этапа. Первый — верификация. После регистрации в сервисе пользователь получает уведомление, позволяющее подтвердить его адрес электронной почты. После одобрения адреса пользователь получает письмо, подтверждающее его членство. Затем его по желанию подписывают на почтовую и/или СМС-рассылку.

Первый шаг в событийно-ориентированном потоке задач — отправка проверочного письма. Чтобы обеспечить его надежную реализацию, нужно сопоставить потенциальных пользователей одной из географических зон. Это позволит продолжать принимать новых пользователей даже в условиях частичного отказа подсистемы регистрации. Каждый шард отправляет конечным пользователям проверочные письма. Первый этап потока завершен. Схема первого этапа приводится на рис. 11.8.

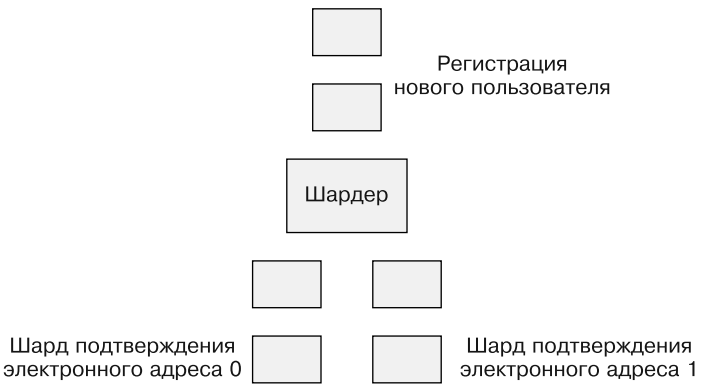

Рис. 11.8. Первый этап потока задач по регистрации нового пользователя

Поток возобновляется при получении подтверждения от пользователя. Оно становится событием в отдельном (но связанном) потоке задач, который отправляет подтверждения и настраивает уведомления. Первый его этап служит примером реализации

паттерна Copier, распределяющего пользователей в две очереди. Первая очередь задач отвечает за отправку приветственных электронных писем, а вторая — за настройку уведомлений.

Как только задачи были распределены в две очереди, очередь на отправку письма отправляет сообщение, и на этом данная ветвь потока завершается. Но за счет использования паттерна Copier активна еще одна ветвь потока задач. Она связана с обработкой настроек уведомлений. Данная очередь задач относится к контейнеру-фильтру, который делит ее на очереди подписки на почтовые и СМС-уведомления. Соответствующие очереди задач подписывают пользователей на уведомления по электронной почте и/или СМС.

Оставшаяся часть потока задач приводится на рис. 11.9.

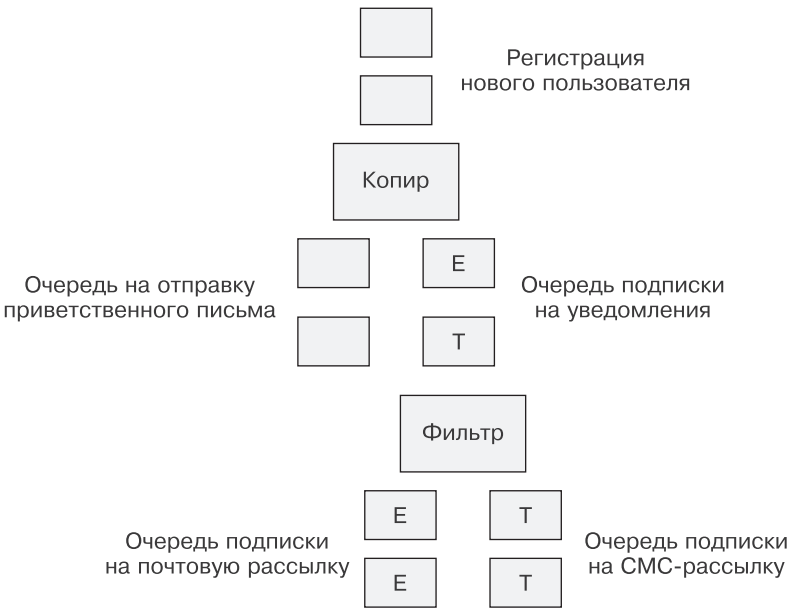

Рис. 11.9. Очередь задач по уведомлению пользователей и рассылке приветственных электронных писем

Инфраструктура publish/subscribe

Мы рассмотрели много разных паттернов, связывающих реализации паттернов событийно-ориентированной пакетной обработки. При фактической реализации подобной системы приходится решать, каким образом управлять потоком данных, идущим параллельно потоку задач. Проще всего будет записывать каждый элемент очереди задач в определенном каталоге локальной файловой системы, за появлением новых задач в котором выполняется наблюдение на конкретном этапе обработки. Использование локальной файловой системы ограничивает рабочую область одним узлом. Можно воспользоваться сетевой файловой системой, распределяющей файлы между множеством узлов, но это усложняет и код приложения, и процесс его развертывания.

При реализации подобных потоков задач принято использовать API или сервис типа pub/sub (publish/subscribe, публикация/подписка). API pub/sub позволяет определить набор очередей, иногда называемых *темами*. Один *издатель* или более публикуют в этих очередях сообщения. По аналогии один *подписчик* или более «прослушивают» их в ожидании новых сообщений. После опубликования сообщение надежно хранится в очереди и таким же надежным образом доставляется подписчикам.

На сегодняшний день большинство публичных облачных провайдеров предоставляют pub/sub-API, к примеру Azure EventGrid или Amazon Simple Queue. Кроме того, на собственном оборудовании или в облачных виртуальных машинах можно использовать довольно популярную реализацию pub/sub-API под названием Apache Kafka (https://kafka.apache.org/). В оставшейся части обзора pub/sub-API в примерах мы будем использовать Kafka, но их будет несложно перенести на другие реализации pub/sub.

Практикум. Развертывание Kafka

Очевидно, существует множество способов развертывания Kafka. Один из простейших подходов — использование контейнера под управлением оркестратора Kubernetes и менеджера пакетов helm.

Helm — менеджер пакетов для Kubernetes, упрощающий развертывание и управление готовыми приложениями наподобие Kafka. Если у вас еще не установлен инструмент командной строки helm, загрузить его можно с сайта http://helm.sh.

Инструмент helm после установки необходимо инициализировать. В процессе инициализации helm разворачивает в вашем кластере компонент под названием tiller и устанавливает ряд паттернов на локальной системе.

```
helm init
```

После инициализации helm установите Kafka, выполнив следующие команды:

```
helm repo add incubator
  http://storage.googleapis.com/kubernetes-charts-incubator
helm install --name kafka-service incubator/kafka
```

Helm-паттерны различаются по уровню поддержки и по уровню готовности к промышленному использованию. Стабильные (stable) паттерны подвергаются строжайшему отбору и поддерживаются официально. Инкубаторные (incubator) паттерны носят более экспериментальный характер и в меньшей степени опробованы в «боевых» условиях. Инкубаторные паттерны полезны для быстрой PoC-реализации (proof-of-concept) сервисов, а также в качестве начальной точки для развертывания рабочей среды сервисов, работающих в рамках Kubernetes.

После установки и запуска Kafka можно создавать темы, а в них — делать публикации. В рамках пакетной обработки тема соответствует выходным данным одного из модулей потока задач. Они, скорее всего, станут входными данными другого модуля в потоке задач.

К примеру, в рамках ранее рассмотренного паттерна Sharder каждому из выходных шардов будет соответствовать своя тема. Если выходной шард называется Photos и всего у вас их три, то и темы также будет три — Photos-1, Photos-2 и Photos-3. После вычисления значения шардирующей функции модуль шардирования публикует сообщение в соответствующей теме.

Рассмотрим, как создать тему. Чтобы получить доступ к Kafka, сначала создадим контейнер в кластере:

```
for x in 012; do
  kubectl run kafka --image=solsson/kafka:0.11.0.0
  --rm --attach --command -- \
    ./bin/kafka-topics.sh --create --zookeeper
    kafka-service-zookeeper:2181 \
      --replication-factor 3 --partitions 10
        --topic photos-$x done
```

Помимо названия темы и ссылки на сервис ZooKeeper, обратите внимание на два интересных параметра: `--replication-factor` и `--partitions`. Множитель репликации устанавливает, на сколько машин будет реплицироваться тема. Он характеризует степень избыточности, доступной в случае отказа. Рекомендуется использовать значение `3` или `5`. Второй параметр — количество разделов в теме. Количество разделов соответствует максимальному числу машин, на которые тема будет распространяться с целью балансирования нагрузки. Поскольку в данном случае мы задали десять разделов, для балансирования нагрузки может использоваться не более десяти копий.

В созданную только что тему теперь можно отправлять сообщения:

```
kubectl run kafka-producer --image=solsson/kafka:0.11.0.0
--rm -it --command -- \
    ./bin/kafka-console-producer.sh
       --broker-list kafka-service-kafka:9092 \
  --topic photos-1
```

После выполнения данной команды появится приглашение командной строки Kafka, откуда можно отправлять сообщения в тему (-ы). Для получения сообщений выполним команду:

```
kubectl run kafka-consumer --image=solsson/kafka:0.11.0.0
--rm -it --command -- \
    ./bin/kafka-console-consumer.sh --bootstrap-server
    kafka-service-kafka:9092\
    --topic photos-1 \
        --from-beginning
```

Запуск этих команд позволяет лишь поверхностно ознакомиться с механизмами коммуникации на основе Kafka-сообщений. Чтобы построить настоящую событийно-ориентированную систему пакетной обработки, вам, скорее всего, придется использовать настоящий язык программирования и инструментарий разработчика Kafka SDK. С другой стороны, не следует недооценивать мощь хорошего bash-сценария!

Этот пример показал вам, как установить инструментарий Kafka в Kubernetes-кластер, и то, насколько сильно он упрощает построение систем на основе очередей задач.

12 Координированная пакетная обработка

В предыдущей главе мы рассмотрели паттерны разделения и сцепления очередей, позволяющие реализовать более сложную пакетную обработку. Дублирование и порождение нескольких наборов выходных данных — значительная часть пакетной обработки, но иногда не менее важным оказывается слияние нескольких выходных потоков данных с целью получения некоторого агрегированного результата. Общая схема подобного паттерна приводится на рис. 12.1.

Наиболее типичный пример подобной агрегации — паттерн MapReduce. Нетрудно заметить, что шаг *Map* соответствует шардированию очереди задач, а шаг *Reduce* — координированной обработке, в результате которой большое количество данных агрегируется в один ответ. Кроме MapReduce, существует еще несколько паттернов пакетной обработки. В этой главе рассмотрены некоторые из них, а также соответствующие приложения.

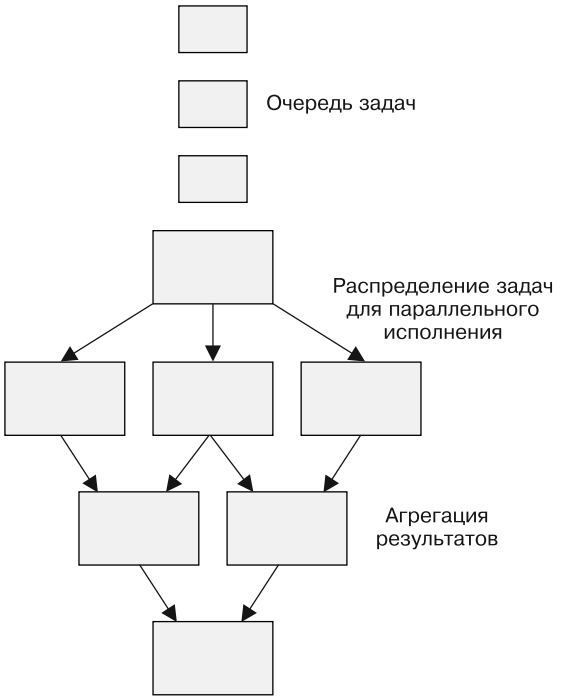

Рис. 12.1. Обобщенная пакетная система распределения задач и агрегации результатов

Паттерн Join (барьерная синхронизация)

В предыдущих главах мы рассмотрели паттерны распределения работы между несколькими вычислительными узлами. В частности, разобрались, как шардированная очередь задач может параллельно распределять нагрузку на несколько шардов очереди задач. Иногда очередной этап обработки потока задач требует наличия полного набора данных, прежде чем можно будет продолжить вычисления.

Одним из вариантов, как показано в предыдущей главе, будет слияние нескольких очередей воедино. Однако слияние попро-

сту объединяет выходы двух очередей в один поток, который подвергнется дальнейшей обработке. Паттерна слияния в некоторых случаях достаточно, но он не гарантирует готовность всего набора данных до момента начала обработки. Следовательно, полнота выполняемой обработки не может быть гарантирована. Кроме того, нет возможности подсчитать агрегированную статистику по обработанным элементам данных.

Нам нужен другой примитив пакетной обработки данных, более строгий и координирующий. Таким примитивом и выступает паттерн Join. Паттерн Join по смыслу аналогичен слиянию потоков. Основная идея данного паттерна состоит в том, что, хотя значительная часть обработки осуществляется параллельно, элементы очереди задач не могут выйти из блока Join, пока не будут вычислены все элементы параллельно вычисляемого набора данных. Подобный прием в параллельном программировании также известен под названием *«барьерная синхронизация»*. Паттерн координированной пакетной обработки Join изображен на рис. 12.2.

Координация с помощью Join гарантирует, что до выполнения агрегации (например, суммирования элементов) ни один элемент данных не останется невычисленным. Достоинство паттерна Join в том, что он обеспечивает наличие всех данных в агрегируемом наборе. Недостаток паттерна Join в том, что перед началом вычислений он требует, чтобы все данные были обработаны на предыдущем этапе вычислений. Это снижает доступный в рамках пакетной обработки уровень параллелизма, а значит, увеличивает задержку выполнения потока задач.

Паттерн Reduce

Как уже было сказано, шардирование может выступать примером реализации фазы *Map* в каноническом алгоритме *Map/Reduce*. Следовательно, нам осталось лишь реализовать фазу *Reduce*. Reduce является примером паттерна координированной

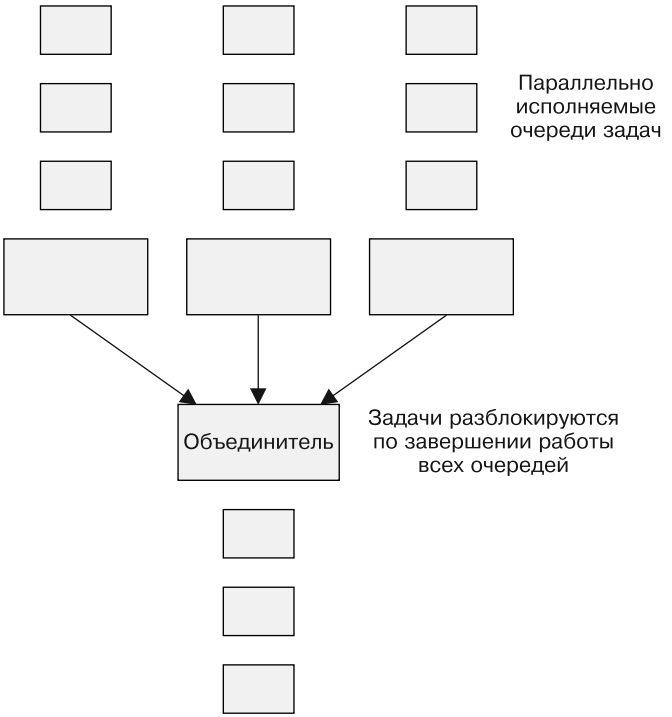

Рис. 12.2. Паттерн пакетной обработки Join

пакетной обработки, поскольку может существовать независимо от того, как поделен входной поток, и использоваться аналогично паттерну Join, то есть для слияния параллельно вычисляемых результатов пакетной обработки элементов данных.

Однако, в отличие от рассмотренного ранее паттерна Join, задача паттерна Reduce — выполнить оптимистичное слияние всех параллельно вычисленных элементов данных в единое исчерпывающее представление исходного множества.

В паттерне Reduce каждый шаг вычислений приводит к сворачиванию нескольких элементов выходных данных в один. Эта фаза называется *сверткой*, поскольку в ней уменьшается объем

выходных данных. Иными словами, исходный набор данных сворачивается до некоторого репрезентативного набора данных, позволяющего найти результат конкретных пакетных вычислений.

Поскольку фаза свертки работает над некоторым участком входных данных и порождает похожие на них выходные данные, ее можно повторять столько раз, сколько необходимо, до тех пор, пока на выходе не получится единственное значение, соответствующее всему набору данных. Это выгодно отличает паттерн Reduce от паттерна Join, поскольку фаза свертки может запускаться параллельно фазе шардирования других участков данных. Для получения конечного результата, конечно же, в итоге придется обработать все данные, но возможность начать обработку раньше позволяет в целом быстрее завершить вычисления.

Практикум. Подсчет

Чтобы понять, как работает паттерн Reduce, рассмотрим задачу подсчета количества вхождений определенного слова в книге. Сначала воспользуемся шардированием, чтобы разделить задачу подсчета на несколько очередей. Можно, например, создать десять разных шардированных очередей, за подсчет слов в каждой из которых отвечает один исполнитель. Книгу можно шардировать между этими десятью очередями по номеру страницы. Страницы с номером, заканчивающимся на 1, уйдут в первую очередь, на 2 — во вторую и т. д.

Как только все исполнители закончат подсчет на своих страницах, каждый из них запишет результат на листочке бумаги. Например:

```
a: 50
the: 17
cat: 2
airplane: 1
...
```

Эти данные передаются на фазу свертки. Напомню, что паттерн Reduce выполняет свертку путем комбинации двух и более элементов входных данных в один элемент выходных.

Второй набор выходных данных:

```
a: 30
the: 25
dog: 4
airplane: 2
...
```

Далее в процессе свертки количество экземпляров слов в каждом из шардов суммируется:

```
a: 80
the: 42
dog: 4
cat: 2
airplane: 3
...
```

Очевидно, что каждая последующая свертка выполняется над выходными данными предыдущей, и так до тех пор, пока не останется единственный элемент выходных данных. Ценность этого факта в том, что свертки могут выполняться параллельно.

Таким образом, вы видите, что результатом свертки будет единственный элемент выходных данных.

Суммирование

Суммирование некоторого множества значений — похожая, но немного другая разновидность свертки. Она подобна подсчету, но каждый раз к аккумулятору прибавляется не единица, а значение элемента данных.

К примеру, вы хотите посчитать численность населения Соединенных Штатов. Предположим, для этого вы сначала опре-

делите численность населения в каждом городе, а затем просуммируете полученные результаты.

Первый шаг — разделить задачу на очереди по городам с шардированием по штатам. Это хороший первый шаг, но очевидно, что даже при параллельном распределении задачи одному человеку будет трудно подсчитать численность населения в каждом городе. Следовательно, необходимо углубить шардирование, на этот раз по округам.

На данный момент мы распараллелили задачу по штатам, затем по округам. Очереди задач в каждом округе на выходе выдают поток пар вида `(город, население)`.

Как только на выходе появляются значения, начинает работу паттерн Reduce.

В данном случае ему даже не обязательно знать о двухуровневом шардировании. Ему достаточно взять два выходных элемента, например `(Сиэтл, 4000000)` и `(Нортгемптон, 25000)` и просуммировать их. В результате получится новый выходной элемент `(Сиэтл-Нортгемптон, 4025000)`. Очевидно, что, как и в случае с подсчетом, такая свертка может выполняться неограниченное количество раз, в результате чего получится единственное выходное значение, содержащее общую численность населения США. Опять-таки важно то, что почти все необходимые вычисления происходят параллельно.

Гистограмма

В качестве последнего примера применения паттерна Reduce рассмотрим задачу, в которой одновременно с определением численности населения США путем шардирования и свертки нужно построить модель среднестатистической американской семьи. Для этого желательно получить *гистограмму* размера

семьи, то есть модель, оценивающую общее количество семей с количеством детей от 0 до 10. Многоуровневое шардирование организуется так же, как и прежде (вероятно, даже с использованием тех же исполнителей).

Выходным значением фазы сбора данных в этом случае будет гистограмма по городу:

```
0: 15%
1: 25%
2: 50%
3: 10%
4: 5%
```

Из предыдущих примеров видно, что можно объединить все эти гистограммы в одну, получив тем самым общую картину по США. Сперва может быть довольно трудно понять, как выполнить слияние гистограмм. Взяв данные гистограммы и данные о населении из предыдущего примера, видим, что если умножить данные гистограмм на соответствующую численность жителей, то можем получить количество населения для каждого элемента данных объединенной гистограммы. Поделив эти значения на сумму численностей населения, соответствующих объединяемым гистограммам, получим данные для объединенной гистограммы. Таким образом, можно применять паттерн Reduce столько раз, сколько нужно, пока не получится единственная гистограмма.

Практикум. Конвейерная разметка и обработка изображений

Чтобы понять, как координированная пакетная обработка может быть использована для выполнения сложных пакетных задач, рассмотрим задачу разметки и обработки наборов изображений. Предположим, есть большой набор фотографий

шоссе в час пик. Нужно посчитать количество автомобилей, грузовиков и мотоциклов, а также статистическое распределение цветов машин. Допустим также, что в целях анонимизации предварительно выполняется размытие изображений номерных знаков.

Фотографии предоставляются в виде последовательности URL-адресов HTTPS, каждый из которых указывает на необработанное изображение. Первый этап конвейера — нахождение и размытие номерных знаков. Чтобы упростить задания в очередях, введем двух исполнителей. Один будет обнаруживать номерной знак, а другой — размывать соответствующую область изображения. Объединим эти два контейнера-исполнителя в одну группу, как показано в главе 10 при рассмотрении паттерна Multi-Worker. Такое распределение обязанностей на первый взгляд может показаться избыточным. Его польза в том, что контейнер-исполнитель для размытия фрагментов изображений можно использовать повторно, например для размытия лиц на фотографиях.

Кроме того, для повышения надежности и максимизации параллелизма будем шардировать изображения на несколько очередей. Полная схема потока задач по размытию участков изображений с применением шардирования приведена на рис. 12.3.

После размытия номерных знаков на всех изображениях результат загрузим в другое место, а исходные изображения удалим. Оригиналы не следует удалять до тех пор, пока не будут обработаны *все* изображения. Они понадобятся на случай катастрофического сбоя, если придется заново перезапустить процесс обработки. Для того чтобы дождаться обработки всех изображений, воспользуемся паттерном Join, рассмотренным в предыдущей главе. С его помощью мы объединим шардированные очереди задач в одну общую очередь, которая освободит

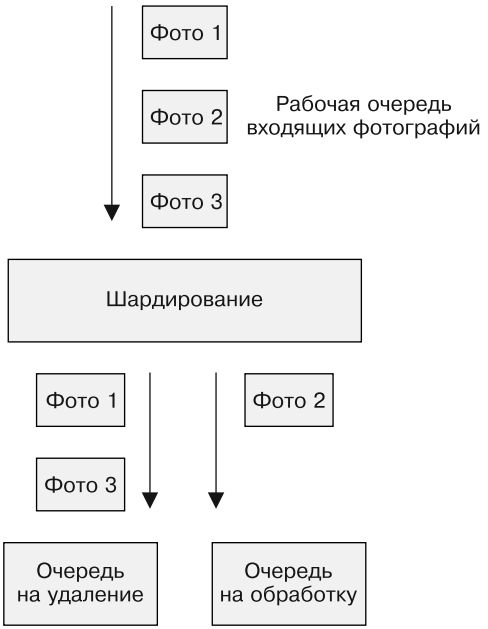

Рис. 12.3. Шардированная очередь с несколькими шардами для размытия изображения

элементы для дальнейшей обработки только после того, как все шарды завершат работу.

Теперь можно удалить оригиналы и начать распознавание марок и цветов машин. Пропускную способность данного конвейера также хотелось бы максимизировать, поэтому воспользуемся паттерном Copier из предыдущей главы и создадим две отдельные очереди:

- очередь задач по удалению оригиналов;
- очередь задач по определению цвета и марки машины.

На рис. 12.4 схематически изображены упомянутые стадии обработки.

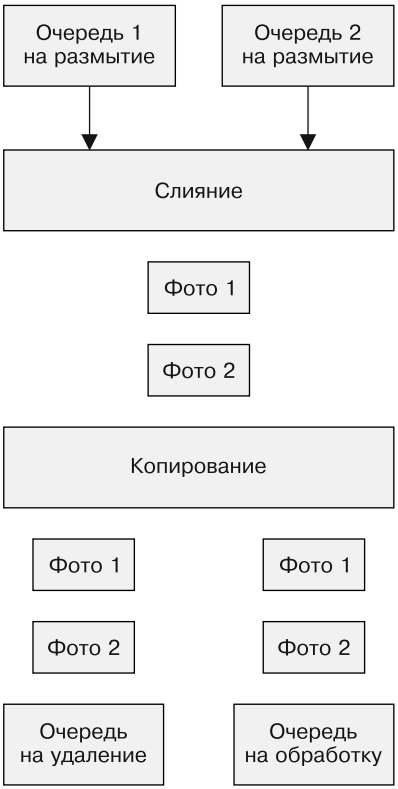

Рис. 12.4. Этапы слияния и копирования очередей, удаления исходных изображений и распознавания образов

Наконец, необходимо разработать очередь задач по распознаванию автомобиля и его цвета, которая бы подводила статистику по данным параметрам. Для этого сначала применим шардирование, чтобы распределить работу на несколько очередей. В каждой из очередей будет два исполнителя: один будет распознавать положение и тип транспортного средства, а второй — определять цвет распознанной области. Для слияния снова воспользуемся паттерном Multi-Worker, рассмотренным в главе 10. Как и ранее, разделение кода на несколько контейнеров

позволяет использовать контейнер, определяющий цвет, в других системах для определения цветов других объектов, а не только автомобилей.

Выходные данные очереди в формате JSON будут выглядеть примерно следующим образом:

```
{
  "ТС": {
    "автомобиль": 12,
    "грузовик": 7,
    "мотоцикл": 4
  },
  "цвета": {
    "белый": 8,
    "черный": 3,
    "синий": 6,
    "красный": 6
  }
}
```

В этих данных представлена информация, найденная в одном изображении. Чтобы собрать данные воедино, воспользуемся описанным ранее паттерном Reduce, ставшим популярным благодаря MapReduce. Просуммируем все элементы точно так же, как и в примере с подсчетом. Стадия свертки в качестве результата выдает итоговое количество образов и цветов, найденных во всем множестве снимков.

13 Заключение — новое начало?

Все компании, независимо от своего происхождения, становятся цифровыми. Такие преобразования требуют создания API и сервисов, используемых в мобильных приложениях, устройствах Интернета вещей (IoT) и даже в автономном транспорте и других системах. Рост ответственности, возлагаемой на такие системы, означает, что они должны проектироваться с учетом избыточности, отказоустойчивости и высокой доступности. В то же время требования бизнеса обуславливают потребность в высокой маневренности в плане разработки и внедрения нового ПО, поддержки старого, а также экспериментов с новыми пользовательскими и программными интерфейсами. Сочетание упомянутых факторов привело к значительному увеличению потребности в распределенных системах.

Создавать такие системы все еще слишком сложно. Суммарная стоимость их разработки, обновления и сопровождения очень

высока. С другой стороны, количество людей, обладающих необходимыми навыками и способностями, слишком мало, чтобы удовлетворить растущий спрос.

Исторически возникновение подобных ситуаций в сфере разработки ПО и технологий приводило к появлению новых уровней абстракции и паттернов проектирования ПО. Они делали процесс разработки быстрее, проще и надежнее. Впервые это произошло с созданием первых компиляторов и языков программирования. Потом появились объектно-ориентированные языки программирования на основе управляемого промежуточного кода. В каждый из этих моментов накопленные технические результаты кристаллизовали суть знаний и навыков экспертов в виде множества алгоритмов и паттернов соответственно. Это позволило пользоваться данными результатами более широкой аудитории разработчиков-практиков. Развитие технологий и укоренение паттернов делало процесс разработки программного обеспечения более демократичным и расширяло круг разработчиков, способных строить приложения на новой платформе. Это, в свою очередь, привело к появлению еще большего количества приложений и росту их разнообразия, что, в свою очередь, повышало спрос на разработчиков с подобными навыками.

Сегодня мы снова находимся на пороге технической трансформации. Спрос на распределенные системы значительно превышает предложение. К счастью, развитие технологий привело к созданию нового инструментария, расширяющего контингент разработчиков таких систем. Недавнее появление контейнеров и систем их оркестрации обеспечило нас инструментами более простого и быстрого развертывания распределенных систем. Если повезет, эти инструменты вкупе с паттернами и практиками, описанными в данной книге, улучшат распределенные системы, создаваемые современными разработчиками, и, что

важнее, сформируют качественно новое сообщество специалистов, способных создавать подобные системы.

Такие паттерны, как Ambassador, Sidecar, шардирование, FaaS, очереди задач, а также многие другие могут сформировать фундамент, на котором будут строиться сегодняшние и будущие распределенные системы. Разработчикам распределенных систем больше не надо создавать свои системы с нуля в одиночку — им нужно совместно работать над обобщенными, повторно используемыми реализациями канонических паттернов, которые сформируют базис для вновь разрабатываемых систем. Это позволит нам удовлетворить спрос на современные, надежные, масштабируемые API и сервисы, а также будет способствовать созданию нового класса приложений и сервисов в будущем.

Об авторе

Брендан Бёрнс (Brendan Burns) — выдающийся инженер фирмы Microsoft, сооснователь open-source-проекта Kubernetes. В Microsoft он трудится над Azure, в частности над контейнерами и DevOps. До Microsoft он работал в Google над Google Cloud Platform, где участвовал в создании таких API, как Deployment Manager (Диспетчер развертываний) и Cloud DNS (Облачный DNS).

До того как Брендан стал работать с облачными вычислениями, он занимался поисковой инфраструктурой Google, в частности минимизацией задержек при индексировании. Получил степень кандидата технических наук в области информатики в Массачусетском университете в Амхерсте по специализации «Робототехника». Живет в Сиэтле со своей женой Робин Сендерс (Robin Sanders), двумя детьми и кошкой, которая своей железной лапой правит в их доме.

Об иллюстрации на обложке

Животное на обложке книги — яванский воробей. Эту птицу ненавидят в дикой природе, но обожают в неволе. Ее латинское наименование — *Padda oryzivora*. Padda означает «заливное поле», метод выращивания риса, а oryza — сорт риса. Следовательно, *Padda oryzivora* буквально переводится как «пожиратель риса с полей». Иногда этих птиц также называют рисовыми воробьями. Фермеры убивают тысячи диких яванских воробьев, иначе те уничтожат их урожай. Они также ловят этих птиц с целью продажи и употребления в пищу.

Несмотря на неравную борьбу, яванские воробьи процветают в Индонезии, на Яве и Бали, а также в Австралии, Мексике и Северной Америке. Их оперение имеет перламутрово-серую окраску, переходящую спереди в бледно-розовый цвет, а ближе к хвосту — в белый. У воробья черная голова и белые щеки. Ноги, перья вокруг глаз и огромный клюв — ярко-розового цвета. Песнь яванского воробья начинается с отдельных нот, похожих на звон колокольчика. Потом она перерастает в сплошную трель со звуками трещотки, высокими и низкими нотами.

Питаются птицы в основном рисом, но не брезгуют мелкими семечками, травинками, насекомыми и цветковыми растениями.

В дикой природе яванские воробьи обычно строят гнезда из сухой травы под крышами зданий, в кустах или на верхушках деревьев. Они откладывают по 3–4 яйца в период с февраля по август.

Их впечатляющее оперение, завораживающие трели и простота в уходе обусловили высокий спрос на них в качестве домашних питомцев. В настоящее время яванские воробьи находятся под охраной. Обеспечиваются все условия, чтобы спрос на них удовлетворялся за счет птиц, разведенных в неволе.

Многие из животных, изображенных на обложках книг издательства O'Reilly, находятся под угрозой исчезновения. Все они важны для нашей природы. Чтобы узнать, как им помочь, посетите сайт animals.oreilly.com.

Изображение на обложке взято из книги *The Royal Natural History* («Королевская естественная история») Ричарда Лидеккера (Richard Lydekker).

Брендан Бёрнс

Распределенные системы. Паттерны проектирования

Перевел с английского *К. Русецкий*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Художественный редактор	*С. Заматевская*
Корректоры	*Е. Павлович, Т. Радецкая*
Верстка	*К. Подольцева-Шабович*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева,
д. 121/3, к. 214, тел./факс: 208 80 01.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 24.01.19. Формат 60×90/16. Бумага офсетная. Усл. п. л. 14,000.
Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87