O'REILLY

Второе издание

Распределенные системы

Паттерны и парадигмы для масштабируемых и надежных систем на основе Kubernetes

SPRINT book

Брендан Бёрнс

SECOND EDITION

Designing Distributed Systems

Patterns and Paradigms for Scalable, Reliable Systems Using Kubernetes

Brendan Burns

O'REILLY®

ВТОРОЕ ИЗДАНИЕ

Распределенные системы

Паттерны и парадигмы для масштабируемых и надежных систем на основе Kubernetes

Брендан Бёрнс

SPRINT book 2025

ББК 32.988.02-018
УДК 004.738.2
Б51

Бёрнс Брендан

Б51 Распределенные системы. Паттерны и парадигмы для масштабируемых и надежных систем на основе Kubernetes. 2-е изд. — Астана: «Спринт Бук», 2025. — 256 с.: ил.

ISBN 978-601-12-3152-7

Разработчики распределенных систем стремятся создать их надежными, производительными и качественными, но добиться этого непросто. Набор паттернов проектирования позволяет разработчикам ПО и системным архитекторам говорить на одном языке при описании своих систем и учиться на примерах и практиках, разработанных другими.

Популярность контейнеров и Kubernetes открывает путь к применению основных паттернов распределенных систем и многократно используемых контейнеризированных компонентов. В этом практическом руководстве представлена коллекция паттернов, которые помогут вам создавать свои системы с использованием устоявшихся практик, заимствованных из некоторых наиболее эффективных современных распределенных систем. Применяя эти паттерны, вы сможете сделать свои системы более доступными и эффективными, даже если никогда раньше не создавали распределенных систем.

Автор книги Брендан Бёрнс демонстрирует, как адаптировать существующие паттерны проектирования для создания надежных распределенных приложений. Системные инженеры и разработчики узнают, как эти давно известные паттерны позволяют значительно усовершенствовать и повысить качество их систем.

Полностью обновленное второе издание также включает новые главы, посвященные искусственному интеллекту, его обучению и применению для создания надежных систем.

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

ББК 32.988.02-018
УДК 004.738.2

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

ISBN 978-1098156350 англ.
Authorized Russian translation of the English edition of Designing Distributed Systems, 2E ISBN 978-1098156350 © 2025 Brendan Burns.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.
ISBN 978-601-12-3152-7
© Перевод на русский язык ТОО «Спринт Бук», 2025
© Издание на русском языке, оформление ТОО «Спринт Бук», 2025

https://t.me/IT_Portal

Краткое содержание

ЧАСТЬ I. БАЗОВЫЕ ПОНЯТИЯ

Глава 1. Введение 21

Глава 2. Важные концепции распределенных систем 30

ЧАСТЬ II. ОДНОУЗЛОВЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Глава 3. Паттерн Sidecar 45

Глава 4. Паттерн Ambassador 58

Глава 5. Адаптеры 70

ЧАСТЬ III. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ ОБСЛУЖИВАЮЩИХ СИСТЕМ

Глава 6. Реплицированные сервисы с распределением нагрузки 86

Глава 7. Шардированные сервисы 103

Глава 8. Паттерн Scatter/Gather 121

Глава 9. Функции и событийно-ориентированная обработка 131

Глава 10. Выбор владельца 145

ЧАСТЬ IV. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ СИСТЕМ ПАКЕТНЫХ ВЫЧИСЛЕНИЙ

Глава 11. Системы на основе очередей задач 163

Глава 12. Событийно-ориентированная пакетная обработка 177

Глава 13. Координированная пакетная обработка 196

ЧАСТЬ V. УНИВЕРСАЛЬНЫЕ КОНЦЕПЦИИ

Глава 14. Паттерны мониторинга и наблюдаемости 209

Глава 15. Использование и обслуживание ИИ 225

Глава 16. Распространенные паттерны отказов 235

Оглавление

Предисловие ... 13
Кому стоит прочесть эту книгу ... 13
Зачем я написал эту книгу ... 13
Современный мир распределенных систем ... 14
Как ориентироваться в книге ... 14
Условные обозначения ... 16
Онлайн-ресурсы ... 16
Использование примеров кода ... 17
Благодарности ... 17
От издательства ... 18

ЧАСТЬ I
БАЗОВЫЕ ПОНЯТИЯ

Глава 1. Введение ... 21
Краткая история разработки систем ... 22
Краткая история паттернов проектирования в разработке ПО ... 23
Формализация алгоритмического программирования ... 23
Паттерны в объектно-ориентированном программировании ... 24
Расцвет программного обеспечения с открытым исходным кодом ... 25
Ценность паттернов, практик и компонентов ... 25
Стоя на плечах гигантов ... 26
Общий язык обсуждения подходов к разработке ... 26
Общие повторно используемые компоненты ... 27
Резюме ... 29

Глава 2. Важные концепции распределенных систем ... 30
API и RPC ... 30
Задержка ... 31
Надежность ... 32
Процентили ... 33
Идемпотентность ... 34
Семантика доставки ... 35
Реляционная целостность ... 35

Согласованность данных ... 37
Оркестрация и Kubernetes ... 39
Проверки работоспособности ... 39
Резюме ... 40

ЧАСТЬ II
ОДНОУЗЛОВЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Глава 3. Паттерн Sidecar ... 45
Пример реализации паттерна Sidecar. Добавление возможности HTTPS-соединения к унаследованному сервису ... 46
Динамическая конфигурация с помощью паттерна Sidecar ... 47
Модульные контейнеры приложений ... 49
Практикум. Развертывание контейнера topz ... 50
Создание простейшего PaaS-сервиса с помощью паттерна Sidecar ... 51
Разработка модульных и повторно используемых реализаций паттерна Sidecar ... 53
Параметризованные контейнеры ... 53
Определение API всех контейнеров ... 54
Документирование контейнеров ... 56
Резюме ... 57

Глава 4. Паттерн Ambassador ... 58
Использование паттерна Ambassador для шардирования сервиса ... 59
Практикум. Шардируем Redis-хранилище ... 61
Использование паттерна Ambassador для реализации сервиса-посредника ... 64
Использование паттерна Ambassador для проведения экспериментов и разделения запросов ... 66
Практикум. Реализация 10%-ных экспериментов ... 66
Резюме ... 69

Глава 5. Адаптеры ... 70
Мониторинг ... 71
Практикум. Мониторинг с помощью Prometheus ... 72
Журналирование ... 74
Практикум. Нормализация форматов журналов с помощью fluentd ... 75
Мониторинг работоспособности сервисов ... 77
Практикум. Комплексный мониторинг работоспособности MySQL ... 78
Резюме ... 80

ЧАСТЬ III ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ ОБСЛУЖИВАЮЩИХ СИСТЕМ

Глава 6. Реплицированные сервисы с распределением нагрузки 86
Сервисы без внутреннего состояния 86
Датчики готовности для балансировщика нагрузки 88
Практикум. Создание реплицированного сервиса с помощью Kubernetes 89
Сервисы с закреплением сессий 90
Сервисы с репликацией на уровне приложения 92
Добавляем кэширующий слой 92
Развертывание кэширующего сервера 93
Практикум. Развертывание кэширующей прослойки 95
Расширение возможностей кэширующего слоя 97
Ограничение частоты запросов и защита от атак типа «отказ в обслуживании» (DoS) 97
SSL-мост 99
Практикум. Развертывание nginx и SSL-моста 100
Резюме 102

Глава 7. Шардированные сервисы 103
Шардирование кэша 104
Зачем нужен шардированный кэш 105
Роль кэша в производительности системы 106
Реплицированный и шардированный кэш 108
Практикум. Развертывание реализации паттерна Ambassador и сервиса memcache для организации шардированного кэша 109
Шардирующие функции 113
Выбор ключа 115
Согласованные хеш-функции 116
Практикум. Построение согласованного шардированного прокси-сервера 117
Шардирование реплицированных сервисов 118
Системы с «горячим» шардированием 118
Резюме 120

Глава 8. Паттерн Scatter/Gather ... 121
Scatter/Gather с распределением нагрузки корневым узлом ... 122
Практикум. Распределенный поиск в документах ... 123
Scatter/Gather с шардированием листовых узлов ... 125
Практикум. Шардированный поиск в документах ... 126
Выбор подходящего количества листовых узлов ... 127
Масштабирование Scatter/Gather-систем с учетом надежности и производительности ... 129
Резюме ... 130

Глава 9. Функции и событийно-ориентированная обработка ... 131
Как определить, когда полезен подход FaaS ... 132
Преимущества FaaS ... 132
Проблемы разработки FaaS-систем ... 133
Потребность в фоновой обработке ... 134
Необходимость хранения данных в памяти ... 134
Стоимость постоянного использования запросно-ориентированных вычислений ... 135
Паттерны FaaS ... 136
Паттерн Decorator. Преобразование запроса или ответа ... 136
Практикум. Подстановка значений по умолчанию до обработки запроса ... 138
Обработка событий ... 139
Практикум. Реализация двухфакторной аутентификации ... 140
Событийные конвейеры ... 141
Практикум. Реализация конвейера для регистрации нового пользователя ... 142
Резюме ... 144

Глава 10. Выбор владельца ... 145
Как определить, нужен ли выбор владельца ... 147
Основы процесса выбора владельца ... 149
Практикум. Развертывание etcd ... 150
Реализация блокировок ... 151
Практикум. Реализация блокировок в etcd ... 154
Реализация владения ... 155
Практикум. Реализация аренды в etcd ... 156
Параллельный доступ к данным ... 157
Резюме ... 160

ЧАСТЬ IV
ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ
СИСТЕМ ПАКЕТНЫХ ВЫЧИСЛЕНИЙ

Глава 11. Системы на основе очередей задач ... 163
Система на основе обобщенной очереди задач ... 163
Интерфейс контейнера-источника ... 165
API очереди задач ... 166
Интерфейс контейнера-исполнителя ... 167
Общая инфраструктура очередей задач ... 169
Практикум. Реализация генератора миниатюр видеофайлов ... 171
Динамическое масштабирование исполнителей ... 173
Паттерн Multi-Worker ... 175
Резюме ... 176

Глава 12. Событийно-ориентированная пакетная обработка ... 177
Паттерны событийно-ориентированной обработки ... 178
Паттерн Copier ... 179
Паттерн Filter ... 180
Паттерн Splitter ... 181
Паттерн Sharder ... 182
Паттерн Merger ... 184
Практикум. Создание событийно-ориентированного потока задач для регистрации нового пользователя ... 186
Инфраструктура publish/subscribe ... 188
Практикум. Развертывание Kafka ... 188
Устойчивость и производительность в очередях задач ... 191
Перехват заданий ... 191
Ошибки, приоритеты и повторы ... 192
Резюме ... 195

Глава 13. Координированная пакетная обработка ... 196
Паттерн Join (барьерная синхронизация) ... 197
Паттерн Reduce ... 199
Практикум. Подсчет ... 200
Суммирование ... 201
Гистограмма ... 202
Практикум. Конвейерная маркировка и обработка изображений ... 203
Резюме ... 206

ЧАСТЬ V
УНИВЕРСАЛЬНЫЕ КОНЦЕПЦИИ

Глава 14. Паттерны мониторинга и наблюдаемости ... 209
Основы мониторинга и наблюдаемости ... 209
Журналирование ... 211
Метрики ... 214
Базовый мониторинг запросов ... 216
Расширенный мониторинг запросов ... 218
Оповещение ... 219
Трассировка ... 221
Агрегирование информации ... 223
Резюме ... 224

Глава 15. Использование и обслуживание ИИ ... 225
Основы систем ИИ ... 225
Размещение модели ... 227
Распространение модели ... 228
Разработка с использованием моделей ... 230
Генерация ответа с дополнением результатами поиска ... 231
Тестирование и развертывание ... 232
Резюме ... 234

Глава 16. Распространенные паттерны отказов ... 235
Грохочущее стадо ... 235
Отсутствие ошибок — это ошибка ... 237
Клиентские и ожидаемые ошибки ... 238
Ошибки управления версиями ... 239
Миф о необязательных компонентах ... 240
Ой, мы все стерли ... 241
Проблемы с широтой входных данных ... 244
Обработка устаревших заданий ... 246
Проблема второй системы ... 248
Резюме ... 250

Заключение ... 251

Об авторе ... 253

Иллюстрация на обложке ... 254

Отзывы о книге

Ценное руководство для всех, кто работает с масштабируемыми и надежными системами, особенно в контексте Kubernetes. Бёрнс разъясняет сложные концепции распределенных систем и описывает практические паттерны проектирования, что делает эту книгу бесценной для инженеров, занимающихся созданием надежных систем в современных облачных средах.

Раджив Редди Вишака (Rajeev Reddy Vishaka), руководитель отдела разработки программного обеспечения, Coinbase

Книга «Распределенные системы» Брендана Бёрнса содержит углубленное исследование ключевых концепций распределенных систем, от сервисов без состояния и шардированных сервисов до событийно-ориентированной обработки и наблюдаемости. Рекомендуется SR-инженерам и всем другим инженерам, стремящимся использовать мощь Kubernetes для создания отказоустойчивых и высокопроизводительных инфраструктур.

Свапнил Шевате (Swapnil Shevate), профессионал и пропагандист в области проектирования надежных систем

Отличный источник знаний, рассказывающий простым языком о сложностях распределенных систем. Брендан Бёрнс предлагает практические паттерны и парадигмы проектирования, незаменимые для создания современных облачных приложений.

Лалиткумар Пракашчанд (Lalithkumar Prakashchand), инженер Meta Platforms

Второе издание книги «Распределенные системы» остается отличной книгой, знакомящей разработчиков с архитектурными концепциями, следование которым может добавить устойчивости и эффективности новым и существующим системам. Книга описывает набор простых паттернов, особенно хорошо зарекомендовавших себя при применении в комплексе с Kubernetes, и является отличной отправной точкой для создания более совершенных систем.

Энн Карри (Anne Currie), генеральный директор компании Strategically Green Learning and Development и автор книги Building Green Software

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 веб-поисковиков 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 итоге.

Появившиеся не так давно технологии контейнеров и их оркестраторов фундаментально изменили ландшафт разработки распределенных систем. В наше распоряжение попали объект и интерфейс, которые позволяют выражать базовые паттерны проектирования распределенных систем и компоновать контейнеризированные 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 по несколько раз реализовывать одни и те же системы. В сэкономленное время можно сосредоточиться на разработке ключевых частей приложения.

Как ориентироваться в книге

Книга разделена на пять частей следующим образом.

- *Часть I «Базовые понятия»*. Главы 1–2 вводят понятие распределенной системы и знакомят с некоторыми базовыми концепциями, необходимыми для понимания особенностей конструирования распределенных систем, описанных в части II.
- *Часть II «Одноузловые паттерны проектирования»*. В главах 3–5 обсуждаются повторно используемые паттерны и компоненты, применяемые в пределах одного узла распределенной системы. В частности, рассматриваются паттерны Sidecar, Adapter и Ambassador.

- *Часть III «Паттерны проектирования обслуживающих систем»*. В главах 6–8 рассматриваются многоузловые паттерны, применяемые в постоянно работающих обслуживающих системах, таких как веб-приложения. Обсуждаются паттерны репликации, шардинга и распределения работы. Дополнительно в главах 9 и 10 обсуждаются основные концепции распределенных вычислений, такие как работа с функциями, событийно-управляемое программирование и выбор главного узла.
- *Часть IV «Паттерны проектирования систем пакетных вычислений»*. В главах 11–13 рассматриваются паттерны распределенных систем для широкомасштабной обработки данных, в том числе очереди задач, событийно-ориентированная обработка и согласованные рабочие процессы.
- *Часть V «Универсальные концепции»*. В конце книги рассматривается несколько тем, общих для всех распределенных систем. Глава 14 охватывает журналирование, мониторинг и оповещение приложений. Глава 15 предлагает обзор инфраструктуры ИИ. А глава 16 описывает множество распространенных проблем и ошибок проектирования, возникающих снова и снова при создании распределенных систем.

Если вы опытный разработчик распределенных систем, можете пропустить главы 1 и 2. Однако я рекомендую 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 III и IV 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. Тогда можете их просто пролистать, чтобы получить общее представление, но при этом 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

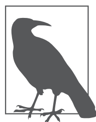
Применяется для отображения URL, адресов электронной почты.

`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.

`Моноширинный наклонный шрифт`

Обозначает текст, который пользователь должен заменить своим значением или значением, определяемым контекстом.



Так обозначается совет, подсказка или общее примечание.

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:

- Docker (https://docker.io);
- Kubernetes (https://kubernetes.io);
- DC/OS (https://dcos.io).

Использование примеров кода

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

Если у вас есть вопросы технического характера или возникла проблема с использованием примеров кода, отправьте электронное письмо по адресу bookquestions@oreilly.com.

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта понадобится разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь 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.

Я также хотел бы поблагодарить технических рецензентов, нашедших время прочитать рукопись, предоставить свои отзывы и сделать эту книгу лучше:

- Динеша Редди Читтибалу (Dinesh Reddy Chittibala);
- Суканью Мурти (Sukanya Moorthy);
- Энн Карри (Anne Currie);
- Лалиткумара Пракашчанда (Lalithkumar Prakashchand);
- Криса Деверса (Chris Devers);
- Уильяма Джамира Сильву (William Jamir Silva);
- Вернера Дийкермана (Werner Dijkerman);
- Раджива Редди Вишаку (Rajeev Reddy Vishaka).

Наконец, я хочу поблагодарить сотрудников издательства O'Reilly и всех, кто поделился отзывами о первом издании этой книги. Вы помогли мне сделать книгу лучше, и я благодарен вам за это.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@sprintbook.kz (издательство SprintBook, компьютерная редакция).

Мы будем рады узнать ваше мнение!

Часть I

Базовые понятия

Прежде чем начать свой рассказ о распределенных системах, я хотел бы поговорить о мотивах и концепциях, формирующих фундамент, на котором основываются распределенные системы. Обо всем этом мы поговорим в первой части, чтобы заложить основу для остального материала книги.

Распределенные системы существуют не в вакууме. Разработка таких систем основана на все усиливающейся роли вычислительных и онлайн-систем в деловой и развлекательной сферах, особенно постоянно действующих критически важных систем, на которые мы полагаемся в повседневной жизни. Кроме того, проектирование современных распределенных систем основано на истории проектирования и конструирования таких систем в прошлом. Эта история, описывающая, как строились системы и, что особенно важно, какие ошибки при этом возникали, привела нас к текущим контейнерным и микросервисным архитектурам, описание которых вы найдете в книге.

Прежде чем изучать особенности проектирования распределенных систем, необходимо познакомиться с основными понятиями серверных систем, а также с фундаментальными концепциями информатики, такими как блокировка и прикладные программные интерфейсы (API). Также нужно получить базовые знания об основных операциях, выполняемых распределенными системами, таких как мониторинг и журналирование. Наконец, поскольку распределенные системы предполагают многочисленные взаимодействия между множеством различных систем, необходимо иметь базовое понимание статистики и знать, как оценить поведение системы, наблюдая за обработкой запросов на разных компьютерах.

После прочтения этих вводных глав у вас должны сформироваться базовые знания о контексте, истории и концепциях, которые помогут вам понять описание устройства этих систем. Эти знания также помогут объяснить, почему некоторые, казалось бы, сложные архитектурные аспекты оказываются необходимыми для надежности или масштабирования.

https://t.me/IT_Portal

ГЛАВА 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, контейнеризированные 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 вычислительные 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 отделены друг от друга.

Постепенно компьютеры 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 годов 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 компьютерах. 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

Это не первый случай, когда подобная трансформация происходит в индустрии разработки 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] в 1962 году. 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

---

[1] *Кнут Д.* Искусство программирования. В 4 т. — М., 2019.

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.

Паттерны в объектно-ориентированном программировании

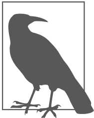
Если появление книг Кнута стало важной вехой в теории компьютерного программирования, то алгоритмы — ключевой составляющей его развития. Однако по мере роста сложности компьютерных программ и увеличения численного состава разрабатывающих их команд с единиц до сотен и тысяч стало ясно, что языков процедурного 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 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» «Паттерны 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.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер.

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 годов. 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, описанных в этой книге, 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 MEAN1) 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. 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 счете 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.

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 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 устройство распределенных систем — лишь часть преимуществ от наличия общего набора паттернов проектирования. Паттерны ценны даже

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

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 поймем, 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* (*sidecar*; будут описаны в главе 3) 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 ценному аспекту 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 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.

Недавняя серия атак на цепочки поставок программного обеспечения показала, что управление зависимостями и обеспечение их безопасности являются важнейшей частью защиты наших приложений. В контексте безопасности цепочек поставок ПО эти совместно используемые компоненты приобретают еще большее значение. Каждая библиотека или приложение, которые мы используем, создают больше зависимостей — и, следовательно, больше рисков. Опора на единую общую реализацию ключевой идеи сокращает общий объем программного обеспечения, от которого должен зависеть мир, а сосредоточение внимания на нескольких зависимостях значительно повышает их защищенность от атак по цепочке поставок ПО.

Резюме

Распределенные системы обеспечивают более высокий 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 паттернов и практик упорядочило и усовершенствовало подходы к алгоритмическому и объектно-ориентированному программированию. Эта книга призвана сделать то же для распределенных систем. Поехали!

https://t.me/IT_Portal

ГЛАВА 2

Важные концепции распределенных систем

Прежде чем углубиться в проектирование распределенных систем, полезно познакомиться с некоторыми ключевыми концепциями, лежащими в основе разработки таких систем. Если у вас уже есть опыт проектирования распределенных систем или систем в целом, то некоторые из этих идей могут оказаться для вас знакомыми и вы можете пропустить их, но надеюсь, что по завершении этой главы вы разберетесь в основах, знание которых необходимо для изучения паттернов и проектов, представленных в остальной части книги.

API и RPC

Прикладные программные интерфейсы (application programming interface, API) формируют ядро любой распределенной системы. Вы могли познакомиться с понятием API в контексте библиотек языков программирования или вызовов операционной системы. Распределенные системы тоже имеют API, только вызовы выполняются по сети между удаленными сервисами. Эти вызовы называются RPC (remote procedure call) — вызовами удаленных процедур. RPC полагаются на базовый сетевой транспорт, обычно (но не только) на протокол HTTP и формат объекта JSON. За эти годы появилось множество других более структурированных протоколов, от CORBA до gRPC, а также протоколов, специфических для баз данных. Хотя такие системы предоставляют дополнительные возможности, выходящие за рамки HTTP + JSON (также называемые REST или RESTful), они продолжают пользоваться большой популярностью из-за их широкой поддержки, простоты и легкости интеграции с веб-интерфейсами.

API могут быть синхронными или асинхронными. Синхронные API возвращают данные только после полной обработки запроса, тогда как асинхронные принимают запрос и возвращают идентификатор операции для отслеживания с помощью последующих запросов на получение статуса. Синхронные API проще в реализации, но продолжительные операции плохо подходят для HTTP или других сетевых протоколов из-за тайм-аутов и других ограничений. Поэтому если длительность операции измеряется минутами, то ее лучше смоделировать как асинхронную.

Как правило, API бесполезны без клиентов, вызывающих их. Если разработчикам придется вручную выполнять HTTP-вызовы к API, то это надежный способ вызвать у них недовольство и получить плохо работающие системы. В небольших системах, обеспечивающих поддержку лишь отдельных языков программирования, такие клиентские библиотеки можно написать вручную. Но более крупные системы полагаются на IDL (Interface Definition Language — язык описания интерфейсов), такие как OpenAPI, позволяющие не только описать структуру API, но и автоматически генерировать программный код клиента практически на любом языке.

Помимо структуры API, основная функциональность интерфейсов также описывается в терминах целевого уровня обслуживания (service level objectives, SLO) — рабочих характеристиках API с точки зрения задержек, надежности и пропускной способности (количества запросов в секунду). Наличие формального SLO позволяет клиентам успешно разрабатывать распределенные системы.

Задержка

Одним из важнейших показателей производительности системы является время, необходимое для выполнения какого-либо действия. Этот показатель обычно называют *задержкой*. Чаще всего задержка измеряется в миллисекундах, реже в микросекундах (очень маленькая задержка) или секундах (очень большая задержка). При измерении задержки важно отразить фактический опыт конечного пользователя (см. раздел «Процентили» далее) и выяснить причины, ее порождающие. Не зная причин, вы будете наблюдать медленную работу вашей системы, но не сможете предложить решений по ее улучшению.

Надежность

Кроме задержки, еще одной важной характеристикой для любой распределенной системы является надежность. В простейшем случае надежность — это отношение числа успешно обработанных запросов к их общему числу. Но, как это часто бывает в инженерных задачах, дьявол кроется в деталях. Возьмем для примера запросы, основанные на протоколе HTTP(s). При использовании этого протокола можно без труда измерить количество неудачных запросов (то есть запросов, которые фактически не были обработаны с точки зрения протокола HTTP). Однако практически в любом веб-сервисе это число стремится к нулю. Причина в том, что HTTP имеет богатую семантику ошибок в виде кода ответа HTTP. Соответственно, почти каждый HTTP-запрос успешно выполняется на уровне протокола, но нередко код ответа указывает на ошибку. Из этого ясно, что характеристика надежности должна измеряться не на уровне протокола, а на уровне кода ответа. Однако HTTP определяет большое количество кодов ответа. Какие из них следует рассматривать как ошибки? Очевидно, что настоящими ошибками являются коды 50x. Большинство из них представляют либо внутреннюю ошибку сервера (500), либо ошибку сетевой передачи (например, 502). Также совершенно очевидно, что коды 20x представляют успешные запросы. К сожалению, между 20x и 50x находится серая зона. Обычно коды 30x (в основном сообщающие о перенаправлении) считаются успешными, хотя в некоторых условиях чрезмерное количество перенаправлений может считаться ошибкой. Но наибольшая неопределенность связана с кодами 40x. Код 404 («не найдено») может говорить об ошибке на стороне клиента, но точно так же он может быть обусловлен неправильной настройкой каталога для поиска файлов в вашем сервисе. Аналогично код 403 («неавторизованный доступ») может говорить и об ошибке на стороне клиента, и о нарушениях в работе вашего сервиса авторизации.

Хотя приведенные выше примеры взяты из протокола HTTP, они наглядно демонстрируют, что измерение надежности является не только важной, но и сложной задачей. На практике обработка кода 404 («не найдено») как ошибки клиента может скрыть истинные проблемы надежности, из-за чего сбои в работе вашей системы могут остаться незамеченными. В то же время всплеск ошибок 404 может быть вызван одним неправильно настроенным клиентом, вызывающим шум и рассылку ложных оповещений, донимающих дежурных

инженеров. Мониторинг в такой серой зоне будет зависеть от приложения. Соответственно, анализ ошибок и обнаружение аномалий с применением ИИ становятся критически важными для различения истинных проблем и ложных оповещений.

Процентили

В книге мы часто будем использовать термин «процентили». Например, 99-й процентиль задержки. Процентили являются ключом к пониманию характеристик распределенных систем, которые по определению предназначены для обслуживания большого количества пользователей и большого количества запросов. Учитывая это, мы должны понимать, как работают наши сервисы. По умолчанию для оценки пользовательского опыта используются средние значения. В этом есть определенный смысл, так как важно понимать, что испытывает средний пользователь. К сожалению, средние значения нередко дают искаженное представление о системе. Тому есть несколько причин, но две самые большие связаны с расчетом среднего значения, а также с особенностями восприятия отдельных метрик (например, задержек в обработке запросов) пользователями. Если говорить о среднем арифметическом, то оно сильно искажается большими или малыми выбросами. Представьте серию измерений задержки: относительно небольшая доля маленьких задержек может исказить среднее значение и заставить вас поверить, что средняя производительность вашей системы намного лучше, чем она есть на самом деле.

Другая причина недостаточной представительности среднего значения заключается в том, что оно рассматривает запросы как независимые друг от друга, однако большинство пользовательских впечатлений складывается из восприятия комбинаций запросов, каждый из которых должен увенчаться успехом. Это означает, что на самом деле некоторого среднего опыта с точки зрения задержки практически не существует, а вероятность того, что каждый пользователь столкнется по крайней мере с одним запросом, обрабатывающимся исключительно медленно, быстро приближается к 100 % с увеличением количества запросов в одной комбинации.

По обеим этим причинам, чтобы по-настоящему понять поведение системы, гораздо полезнее оценить 90-й или 99-й процентиль пользовательского опыта. Каждый из них представляет определенное место

в распределении задержки, то есть когда 90 или 99 % пользователей испытывают меньшую задержку. Например, если 99-й процентиль задержки составляет одну секунду, то это означает, что 99 % пользователей испытывают задержку менее одной секунды. Процентили дают гораздо более точное представление о работе сервиса, но имейте в виду, что даже 99-й процентиль подвержен искажениям при наличии комбинаций из нескольких запросов. В таких случаях 99-й процентиль задержки одного запроса быстро становится 90-м (или даже 50-м) процентилем среди комбинаций при достаточно большом количестве запросов в одной комбинации. Поэтому важно понимать, что именно вы измеряете.

Идемпотентность

Проектирование распределенных систем во многом является изучением вариантов выхода системы из строя. Конечно, никому не хочется, чтобы его системы выходили из строя, и поэтому, чтобы система максимально эффективно *реагировала* на сбой, мы разрабатываем идеи, облегчающие восстановление наших сервисов. Первая из таких идей — *идемпотентность*. Идемпотентность — это свойство операции выдавать один и тот же результат при любом количестве выполнений. Пример идемпотентной операции — операция присваивания $X = 5$. Сколько бы раз ни выполнялась эта операция, она всегда будет давать один и тот же результат: переменная X всегда будет получать значение 5. Сравните ее с операцией $X = X + 1$, которая *не является* идемпотентной. Каждый раз, когда она выполняется, значение X увеличивается на единицу. Очевидно, что конечное значение X зависит от того, сколько раз выполнится эта операция. Если вы знакомы с идеей «инфраструктура как код» и декларативным описанием конфигурации, то знайте, что идемпотентность является прямым аналогом декларативного описания конфигурации.

Идемпотентность — важная концепция для распределенных систем, потому что она значительно упрощает обработку ошибок и сбоев. Если известно, что операция идемпотентна, то, столкнувшись со сбоем, можно просто повторять ее снова и снова, пока она не увенчается успехом. Неважно, сколько раз потребуется выполнить операцию для достижения успеха, потому что мы знаем, что конечный результат всегда будет одинаковым. Идемпотентные операции упрощают проектирование надежных систем.

Семантика доставки

Идемпотентность позволяет нам игнорировать количество выполнений операции, но иногда действительно бывает нужно контролировать, сколько раз конкретный запрос потерпел неудачу.

Обычно доставка сообщения (или успешное выполнение запроса) делится на две категории. Первая: «не менее одного раза». Как следует из названия, семантика «не менее одного раза» гарантирует доставку сообщения или обработку запроса не менее одного раза. Оговорка «не менее» очень важна, потому что при доставке не менее одного раза не дается никаких гарантий относительно того, сколько раз сообщение будет получено.

Альтернативой доставке «не менее одного раза» является доставка «не более одного раза», которая гарантирует, что сообщение никогда не будет доставлено больше одного раза. Как и в случае с доставкой «не менее одного раза», здесь важной частью является оговорка «не более» — семантика «не более одного раза» допускает, что сообщение никогда не будет получено. Хороший пример того, где может пригодиться семантика доставки «не более одного раза», — списание средств с кредитной карты. Очевидно, что несколько одинаковых списаний с одной и той же карты недопустимо. В таких случаях семантика доставки «не более одного раза» предполагает, что в случае сбоя (например, отсутствия оплаты) будет произведена повторная попытка.

Обратите внимание, что в описанных вариантах отсутствует доставка «точно один раз». Это связано с тем, что на практике реализовать семантику доставки «точно один раз» исключительно сложно. Гораздо проще проектировать системы, гарантирующие доставку не менее или не более одного раза.

Реляционная целостность

Причина важности семантики доставки сообщений заключается в том, что она в конечном счете влияет на данные, которые являются основой приложения. Если данные — это банковский счет, то целостность этих данных, очевидно, является очень важной. Обеспечение целостности отдельного элемента данных (например, баланса на вашем банковском счете) играет важную роль, но для распределенных

систем не менее важна также *реляционная целостность*. Под реляционной целостностью понимается целостность или точность данных, хранящихся в нескольких элементах или хранилищах данных в системе. Например, банковский баланс может храниться в виде пары `(user-id, balance)`, а адрес клиента может храниться в другом хранилище тоже в виде пары `(user-id, address)`. Под целостностью этих данных подразумеваются надежные гарантии, что каждому `user-id` в таблице балансов будет соответствовать запись в таблице адресов. Если код приложения полагается на согласованность различных хранилищ данных, то это означает, что он сильно зависит от реляционной целостности. Если, напротив, код может обрабатывать случаи, когда данные в разных хранилищах могут быть не полностью синхронизированы, то это означает, что код не зависит от реляционной целостности.

Почему это важно? Потому что поддержание реляционной целостности требует тщательной синхронизации изменений в нескольких хранилищах данных. Такая синхронизация оказывает существенное влияние на производительность и надежность. В частности, она требует использования распределенных транзакций, охватывающих несколько разных хранилищ, и, соответственно, применения распределенных блокировок, которые делают систему более сложной и менее параллельной.

Различия между поддержкой реляционной целостности и ее отсутствием в общем случае следует рассматривать как различия между хранилищами данных SQL и NoSQL. Традиционные базы данных SQL стремятся сделать все возможное, чтобы обеспечить реляционную согласованность между разными таблицами, предлагая транзакции и семантику отката. Хранилища данных NoSQL, такие как Azure CosmosDB и база данных Cassandra с открытым исходным кодом, реализуют более свободную реляционную семантику, где хранилище просто хранит пары «ключ — значение», а согласованность, если она необходима, обеспечивается прикладным кодом. В целом реляционную согласованность можно рассматривать как компромисс между производительностью и сложностью кода. Если вам не нужна производительность, то согласованность, предлагаемая традиционной базой данных SQL, может значительно упростить ваш код. Напротив, более свободные решения способствуют повышению уровня параллелизма и, соответственно, производительности.

Согласованность данных

Помимо реляционной согласованности, есть еще один важный аспект, связанный с согласованностью, а именно согласованность данных. В любой распределенной системе данные реплицируются (копируются), чтобы можно было обеспечить их доступность в случае сбоев. Репликация данных влияет на производительность, поэтому важно решить, требуют ли данные *строгой* согласованности или достаточно *потенциальной* согласованности.

Очевидно, что согласованность, то есть сохранение одних и тех же данных во всех местах при их изменении, является критически важной стороной хранения данных. Но как быстро измененные данные должны распространиться по системе? Именно это мы имеем в виду, когда говорим о согласованности данных. В строго согласованной системе данные гарантированно копируются повсюду до того, как операция записи данных будет считаться завершенной. В потенциально согласованной системе измененные данные будут скопированы в конечном счете, где в «конечном счете» может затянуться на долгое время.

Чтобы сделать обсуждение чуть более конкретным, рассмотрим систему, которая реплицирует данные в несколько центров обработки данных, разбросанных по всему миру. В строго согласованной системе пользователь изменяет некоторый элемент данных и эти изменения немедленно записываются во все центры обработки данных. И пока все центры не подтвердят, что запись завершена, запрос пользователя на изменение не считается успешным. На первый взгляд такой подход кажется правильным — пока измененные данные не распространятся повсюду, система не должна сообщать пользователю об успехе. Но, к сожалению, этот подход значительно снижает производительность. С ростом количества центров обработки данных растет и количество операций записи, которые требуется выполнить, и может случиться так, что какой-то центр обработки данных начнет работать чрезвычайно медленно или даже окажется недоступен. То есть строгая согласованность обеспечивает надежность данных, но она значительно снижает пропускную способность для приема изменений.

В предыдущем примере система с потенциальной согласованностью может вернуть признак успеха, как только хотя бы один центр

обработки данных подтвердит сохранение изменений. Репликация во все центры обработки данных выполняется асинхронным фоновым процессом, и пользователю не нужно ждать ее завершения. Система говорит: «Изменения приняты», сообщая тем самым, что в конечном счете сделает данные согласованными. Ожидание завершения только одной операции записи делает систему и более надежной, и более производительной. У пользователя значительно меньше шансов столкнуться с медленным или недоступным центром обработки данных при выполнении операции записи.

К сожалению, во многих случаях потенциальная согласованность создает у конечного пользователя впечатление, что ее использование снижает надежность системы. Причиной является проблема «чтения своих собственных записей». Распределенные системы осуществляют репликацию в несколько центров обработки данных, чтобы обеспечить надежность и производительность. В большинстве случаев запросы одного пользователя поступают в один и тот же центр, но иногда, из-за быстро возрастающих нагрузок, сбоев или других сетевых причин, разные запросы от одного и того же пользователя могут поступать в разные центры обработки данных. В системе с потенциальной согласованностью это означает, что после того, как пользователь что-то записал в систему, последующая операция чтения, из-за перебалансировки направленная в другой центр обработки данных, может не найти недавние изменения. Это может вызвать огорчение у пользователя. Представьте, что вы заказали что-то на сайте интернет-магазина, щелкнули на ссылке «просмотреть мои заказы» и не увидели только что оформленного заказа. Есть вероятность, что вы больше никогда не будете заказывать товары на этом сайте.

Из-за этих сложностей выбор между строгой и потенциальной согласованностью никогда не бывает однозначным. Многое зависит от варианта использования, а также от требований к производительности системы. Но будьте осторожны, выбирая подход для использования в вашей системе. Решение о выборе строгой или потенциальной согласованности скажется на всех аспектах архитектуры вашей системы и ее реализации. По завершении реализации будет очень трудно изменить модель хранения, а последствия решения часто проявляются через годы.

Компромиссы между *согласованностью*, *доступностью* и *устойчивостью к сбоям в сети* известны как теорема CAP, которая гласит:

вы сможете построить систему, обладающую двумя из трех характеристик, например согласованную и доступную, но неустойчивую к сбоям в сети, но не сможете построить систему, обладающую всеми тремя характеристиками. Стоит отметить, что теорема CAP не является безапелляционной. Да, вы не сможете построить систему, гарантирующую все три характеристики, но сможете обеспечить разумный уровень для каждой из них. Для некоторых приложений достаточно уровня доступности 99 %, и этот факт можно использовать как компромисс, чтобы обеспечить необходимую согласованность и устойчивость к сбоям в сети. По сути, теорема CAP гласит, что оптимизация одной из характеристик системы фактически ухудшает другие.

Оркестрация и Kubernetes

Современные распределенные системы развертываются не в вакууме или не только с использованием кода, написанного разработчиками системы. Повсеместное распространение распределенных систем привело к созданию общей инфраструктуры поддержки развертывания и эксплуатации распределенных систем. Эту инфраструктуру обычно называют оркестратором, потому что она оркеструет (организует) работу распределенной системы, не требуя реализации в системе какой-то особой логики. Наиболее распространенным оркестратором является Kubernetes — фреймворк с открытым исходным кодом, предназначенный для развертывания контейнерных приложений и управления ими. Оркестратор отвечает за приведение системы в желаемое состояние (например, создает три реплики пользовательского интерфейса и пять реплик серверных компонентов) и поддержание ее в этом состоянии.

Проверки работоспособности

Для успешного управления распределенной системой оркестратор должен иметь возможность оценить работоспособность ее частей. С помощью проверок работоспособности оркестратор определяет, когда приложение готово к передаче балансировщику нагрузки, когда приложение неработоспособно и его необходимо перезапустить или когда развертывание должно быть прервано из-за нарушения работоспособности приложения. Конечно, есть некоторые базовые

показатели работоспособности (например, запущена ли программа), которые оркестратор может наблюдать извне, но для по-настоящему полного понимания состояния приложения требуется, чтобы приложение реализовало полноценные проверки работоспособности.

В отношении информации о работоспособности, передаваемой оркестратору, существует два разных понятия работоспособности. Первое понятие наиболее простое: приложение всего лишь сообщает оркестратору, что оно работает. Приложения, переставшие отвечать на запросы *проверки работоспособности*, должны быть остановлены. И если развертывание вызывает значительное уменьшение количества работающих приложений, то оно должно быть остановлено. Второе понятие — *готовность*. Состояние готовности указывает, что приложение готово к использованию. На первый взгляд готовность и работоспособность кажутся одним и тем же, но нередко работоспособное приложение может быть не готовым к использованию. Самый простой пример — когда приложение только что запущено и инициализирует свое состояние, возможно загружая и кэшируя большие файлы по сети. Такое приложение работоспособно, оно выполняет полезную работу и не должно быть завершено, но оно не готово к использованию, пока не загрузит все нужные файлы и не сохранит их у себя в кэше. Правильно реализуя проверки работоспособности и готовности, можно гарантировать безупречное взаимодействие распределенной системы с оркестратором, под управлением которого они выполняются.

Резюме

В этой главе описан ряд ключевых концепций проектирования и разработки распределенных систем. Эти концепции формируют основу для понимания требований и описания характеристик систем, которые мы создаем. Прежде чем двигаться дальше, обязательно проверьте себя — достаточно ли хорошо вы понимаете их.

Часть II

Одноузловые паттерны проектирования

В книге описываются распределенные системы — приложения, состоящие из множества компонентов, работающих на множестве машин. В этой 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 счете именно группа контейнеров, локализованная на одной машине, представляет собой базовый элемент паттернов проектирования распределенных систем.

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 рассмотрим 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, сервис, который непосредственно взаимодействует с пользователем, и фоновый загрузчик необходимо поместить 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 других причин разместить одноузловое приложение в нескольких контейнерах. 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 выделять 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.

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 расспланировано в рамках одной машины.

Они также подразумевают, что все контейнеры в рамках паттерна смогут 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 (https://kubernetes.io/) — система с открытым исходным кодом для автоматизации развертывания, масштабирования и управления контейнеризированными приложениями. За более полной информацией предлагаю обращаться к моей книге *Kubernetes: Up and Running* (O'Reilly, 2017).

ГЛАВА 3

Паттерн Sidecar

https://t.me/IT_Portal

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. 3.1.

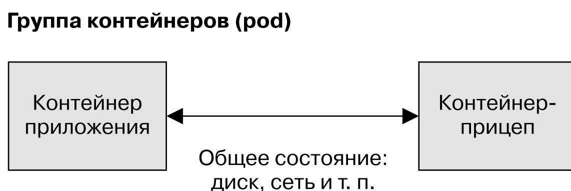


Рис. 3.1. Обобщенный паттерн Sidecar

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, и потому 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.

Контейнеризировать 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 поддержку HTTPS в это приложение — задача куда более сложная. Команде нужно принять решение — или воскресить старую систему сборки, 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. 3.2). 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. Похожую форму этого паттерна можно также использовать для добавления автоматической ротации сертификатов или даже аутентификации и авторизации в унаследованные веб-приложения, которые порой очень непросто изменить.

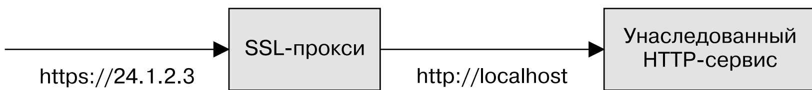


Рис. 3.2. HTTPS-прицеп

Динамическая конфигурация с помощью паттерна Sidecar

Простое проксирование трафика в уже существующее приложение не единственный вариант применения 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 TOML, 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. Именно для этой цели Kubernetes поддерживает ресурсы `ConfigMap`.

По аналогии с примером про HTTPS новые приложения можно писать так, чтобы их конфигурация была динамической и ее можно было получать с помощью облачного API. 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. 3.3, 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.

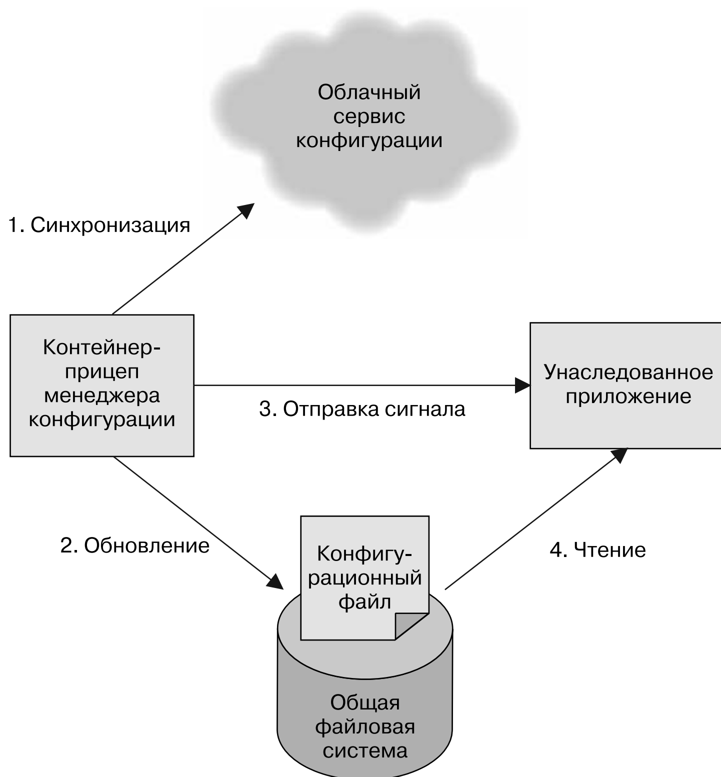


Рис. 3.3. Пример использования паттерна Sidecar для динамического управления конфигурацией

Унаследованное приложение при запуске предсказуемо загружает конфигурацию из файла в файловой системе. Эта конфигурация помещается в файловую систему с помощью тома `ConfigMap`, который размещает текущее содержимое ресурса `ConfigMap` в определенном месте в файловой системе. Менеджер конфигурации при запуске считывает данные конфигурационного API и ищет различия между локальной файловой системой и конфигурацией, сохраненной в API. 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 универсальной надстройки,

которую разработчики смогут добавлять в свои приложения. 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. Другое решение — функциональность `topz` можно развернуть в виде контейнера-прицепа, работающего в том же пространстве идентификаторов процессов, что и контейнер приложения. Такой `topz`-контейнер может осуществлять интроспекцию всех запущенных процессов и предоставлять 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 `topz` в действии, сначала необходимо создать еще один контейнер, который послужит контейнером 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
```

Она запустит контейнер-прицеп topz в том же самом пространстве идентификаторов процессов контейнера приложений. Заметьте, что вам, возможно, придется поменять номер порта, 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 с помощью паттерна Sidecar

Паттерн Sidecar может использоваться не только для адаптации и мониторинга, но и 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 развертывать код на рабочих серверах путем загрузки его в Git-репозиторий. Рассмотрим, как с помощью паттерна Sidecar 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 так, чтобы автоматически перезапускаться 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 легче читать.

Node.js-приложение и прицеп с Git-синхронизатором, реализующие наш простейший PaaS-сервис, развертываются и исполняются совместно на одном узле (рис. 3.4). После развертывания прицеп будет автоматически обновлять файлы в контейнере приложения по мере их загрузки в Git-репозиторий.

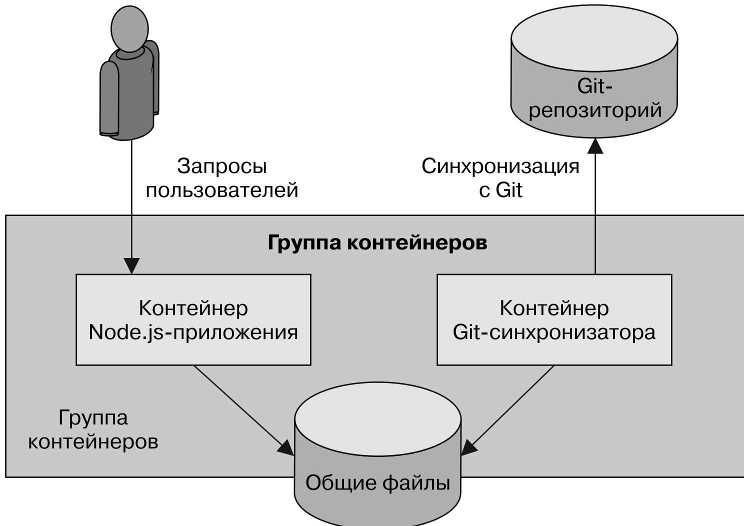


Рис. 3.4. Простейший PaaS-сервис на базе паттерна Sidecar

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 передать 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 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:

```
...
# Основной веб-сервер прослушивает 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 схемы Open Containers Initiative (https://oreil.ly/8FNA2). Open Containers Initiative (OCI) — это орган, управляющий спецификацией, которая определяет, что такое образ контейнера. Частью этой спецификации является определение общего набора аннотаций/меток. Используя общую таксономию 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 вы 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.

ГЛАВА 4

Паттерн Ambassador

В предыдущей главе мы рассмотрели паттерн Sidecar, в рамках которого контейнер-прицеп функционально дополняет существующий контейнер приложения. В этой главе вы познакомитесь 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. 4.1.

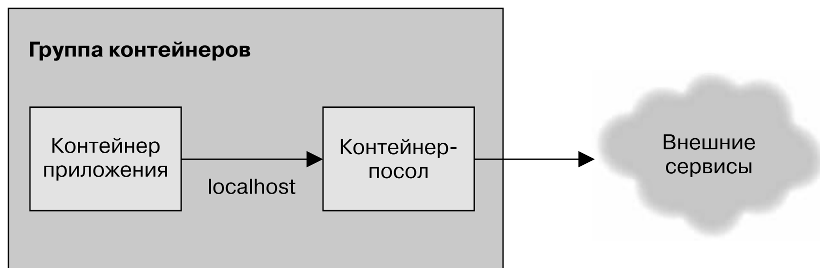


Рис. 4.1. Обобщенный паттерн Ambassador

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 контейнеризированный 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 7.

Схема шардированного сервиса представлена на рис. 4.2.

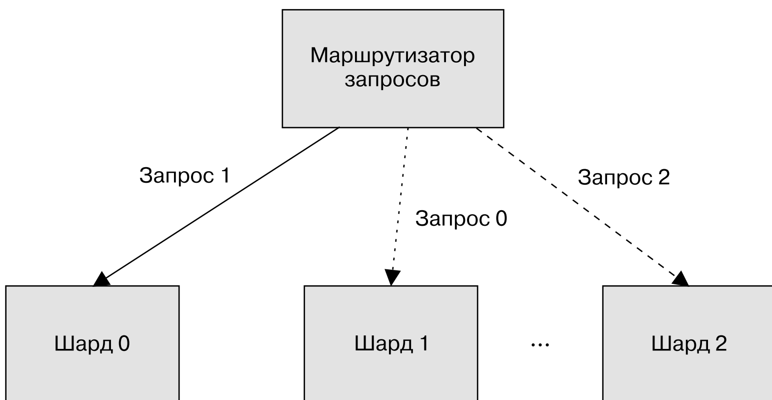


Рис. 4.2. Обобщенная схема шардированного сервиса

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, распределенной 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, выбор подхода зависит от конкретных особенностей вашего конкретного приложения. 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 как кэш или долговременное хранилище данных. В этом примере мы задействуем его в качестве кэша. Начнем с развертывания шардированного Redis в кластере Kubernetes. Для этого обратимся к API `StatefulSet`, поскольку он дает каждому шарду уникальное DNS-имя, которыми мы воспользуемся при настройке прокси.

`StatefulSet` для Redis будет выглядеть следующим образом:

```
apiVersion: apps/v1
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` и разверните его 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]
```

Конечно, недостаточно просто запустить несколько копий Redis — нам также необходимы имена, по которым к ним можно обращаться. Воспользуемся для этого Kubernetes-ресурсом `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-service.yaml` и разверните его командой `kubectl create -f redis-service.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://oreil.ly/IFxFA). 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 этой конфигурации 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 мы можем развернуть нашу реализацию паттерна Ambassador. Определим 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. 4.3.

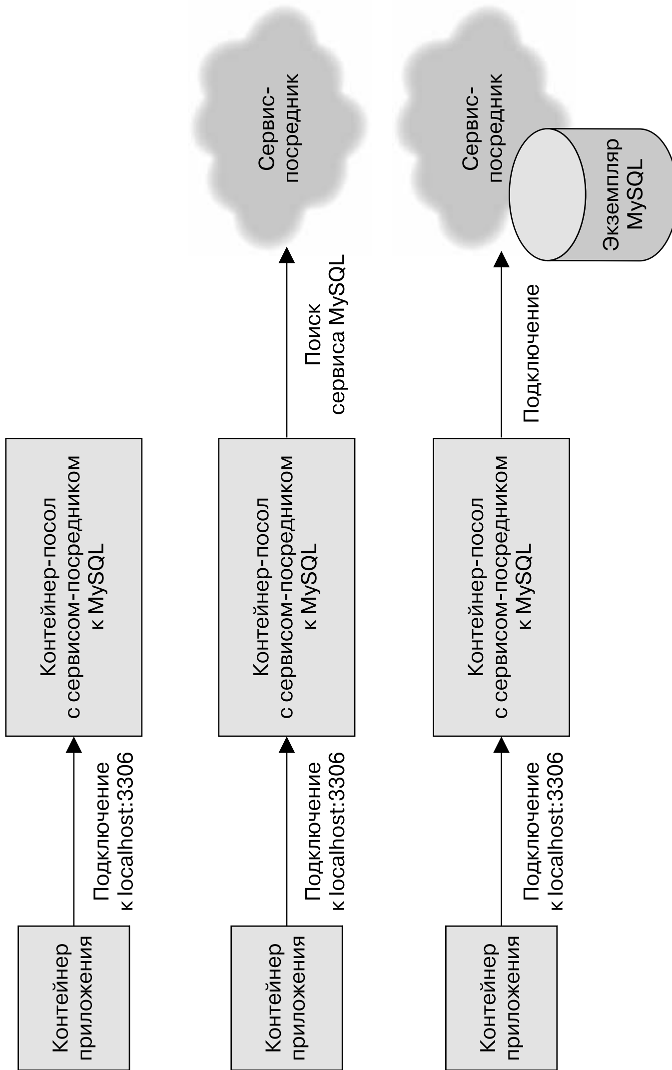


Рис. 4.3. Контейнер-посол сервиса-посредника создает экземпляр MySQL-сервиса

Использование паттерна Ambassador для проведения экспериментов и разделения запросов

Последний пример применения 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 эту возможность используют для экспериментов с новыми или бета-версиями сервиса, чтобы определить степень надежности и производительности новой версии по сравнению с существующей.

Кроме того, разделение запросов иногда используется для дублирования или деления трафика таким образом, что он распределяется как в рабочую версию ПО, так и в 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 очевидно, 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

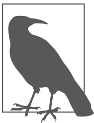
применения в контейнере-после, воспользуемся следующей конфигурацией (обратите внимание, что она рассчитана на HTTP, но ее легко адаптировать под HTTPS):

```
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. Это необходимо, чтобы пользователь не попадал попеременно то в основную, то в тестовую версию сервиса. Благодаря этому пользователи будут взаимодействовать с приложением единообразно.

Параметр `weight` указывает, что 90 % трафика должно приходиться на основное приложение, а 10 % — на тестовую версию.

Как и в других примерах, развернем данную конфигурацию как объект ConfigMap в Kubernetes.

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

Она предполагает, 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 меткам
    app: web
```

Затем развернем 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.

Резюме

Паттерн Ambassador очень помогает разработчикам приложений, давая возможность инкапсулировать сложную логику, необходимую для масштабирования или надежности, например шардинг, и предоставлять интерфейс, упрощающий использование сложной системы. Для инженеров платформ контейнеры-послы могут стать ценным инструментом, помогающим создавать мощные и простые в использовании платформы.

ГЛАВА 5

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 обеспечить вывод файлов журнала только в `stdout`, а также соблюдение 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 журналирования, 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 таком случае 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. Схема на рис. 5.1 обобщенно иллюстрирует данный паттерн.

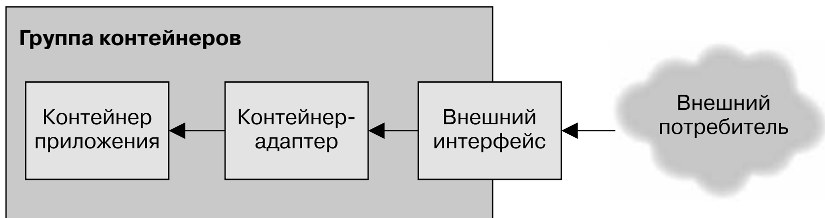


Рис. 5.1. Обобщенный паттерн Adapter

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 самописный код, 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 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, несовместимом

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 объединить существующий контейнер с Redis и адаптер, осуществляющий преобразование в формат Prometheus, чтобы в результате получить Redis-сервер с поддержкой мониторинга

при минимуме усилий с нашей стороны. Развертывание такой же функциональности без применения паттерна Adapter потребовало бы гораздо больше усилий и привело бы к менее управляемому результату, поскольку любое обновление Redis или адаптера требует дополнительных трудозатрат на обновление.

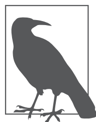
Журналирование

Как и в случае с мониторингом, системы очень неоднородно журналируют данные. Они 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 контейнеризированных 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 журналирования. 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 — 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 изменить сам контейнер приложения? Если вы разработчик, отвечающий за контейнер приложения, то это может оказаться хорошим решением. Адаптация самого приложения или его контейнера путем реализации 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://www.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

приложений. К сожалению, инструмент `SLOWLOG` доступен только в виде команды сервера Redis, а это означает, что такие проблемы сложно отлаживать ретроспективно, когда нет возможности сделать это сразу. Чтобы преодолеть это ограничение, можно воспользоваться сервисом `fluentd` и паттерном Adapter, добавив 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. Для слежения за медленными запросами также воспользуемся плагином `fluent-plugin-redis-slowlog` (https://oreil.ly/yvC6O). Сконфигурировать его 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.

Аналогично можно организовать мониторинг журналов в системе Apache Storm (https://storm.apache.org/). Storm предоставляет данные посредством RESTful API, что само по себе удобно, но имеет ограничения: в момент возникновения проблемы мы не наблюдаем за системой. Как и в случае с Redis, можно воспользоваться `fluentd`-адаптером и преобразовать данные Storm в упорядоченный по времени журнал, к которому можно осуществлять запросы. Чтобы добиться этого, можно развернуть `fluentd`-адаптер с развернутым в нем плагином `fluent-plugin-storm`.

Плагин стоит сконфигурировать так, чтобы он указывал на сервер localhost, поскольку опять-таки мы работаем с группой контейнеров с общим сетевым пространством. Конфигурационный файл плагина будет выглядеть так:

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

Этот адаптер создает мост между Storm и временной последовательностью журнальных записей.

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, — с помощью контейнера-адаптера. 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 привлекательнее, чем 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 код 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 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.

Резюме

Иногда мир слишком далек от идеала. Компоненты, необходимые нам для построения систем, могут иметь разные интерфейсы или использовать разные протоколы, например, для журналирования, мониторинга и вызова удаленных процедур (RPC). Адаптеры в таких случаях могут оказаться чрезвычайно ценным инструментом, помогающим добавить согласованность в наши системы и позволяющим всем компонентам использовать согласованный протокол и способ подключения. Часто реализовать паттерн Adapter оказывается значительно проще, чем пытаться изменить основной проект.

Часть III

Паттерны проектирования обслуживающих систем

В предыдущей главе мы рассмотрели паттерны группировки 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.

С недавних пор плотно вошел в обиход термин *«микросервисы»*, описывающий 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. III.1 и III.2.

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

---

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

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.

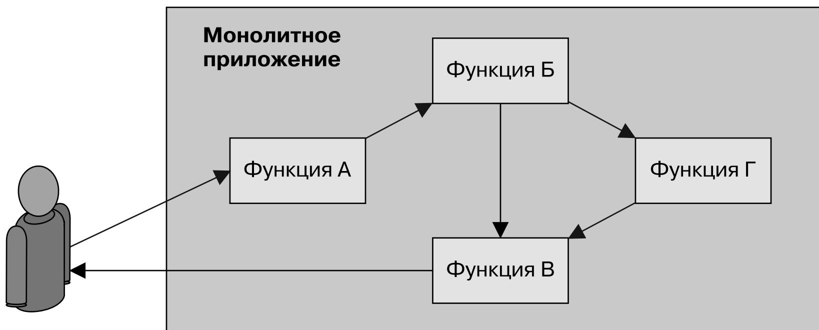


Рис. III.1. Монолитный сервис, вся функциональность которого сосредоточена в одном контейнере

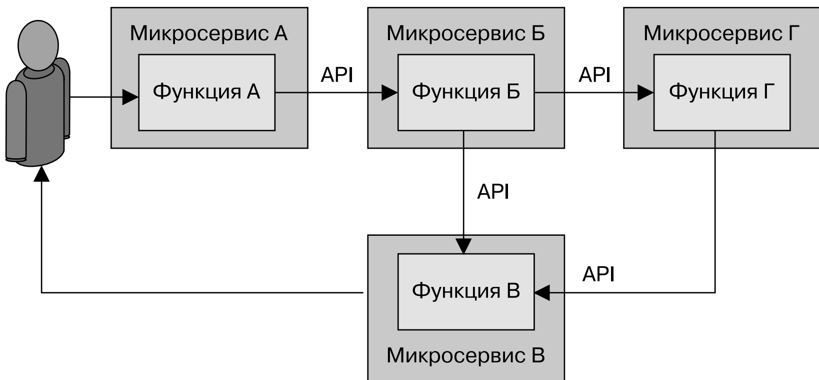


Рис. III.2. Микросервисная архитектура, в которой под каждую функцию выделяется отдельный контейнер

Проектирование архитектуры сервиса во многом напоминает проектирование иерархии классов. Основная цель — обеспечить абстрагирование, инкапсуляцию и модульность. Важно отметить, что в каждой из этих категорий «правильная» конструкция имеет такое же отношение к людям, участвующим в создании проекта, как и к разрабатываемому основному приложению.

Абстрагирование и инкапсуляция часто можно рассматривать как что-то взаимодополняющее друг друга. Абстракция позволяет не беспокоиться о деталях реализации сложной функции и сосредоточиться на том, что с ее помощью можно сделать. Программная модель машинного обучения, распознающая объекты на изображении, имеет довольно сложное устройство, но абстрактный сервис `findObjectsInImage(image): objects` прост в использовании. Распределенные системы тоже очень сложны, и попытка удержать все детали в голове быстро приводит к информационной перегрузке. Абстрагирование позволяет командам сосредоточиться на возможностях высокого уровня, а не на деталях реализации низкого уровня.

Инкапсуляция — это то же самое, но наоборот; инкапсуляция позволяет командам предоставлять услугу, скрывая детали реализации от пользователя. Сокрытие деталей имеет решающее значение, поскольку дает возможность команде разработчиков вносить изменения, не нарушая работу своих пользователей. Это делает команды более гибкими, поскольку увеличивает их независимость друг от друга. Вместе инкапсуляция и абстракция позволяют создавать команды оптимального размера для разработки и поставки. Достаточно большие, чтобы решать сложные задачи, но достаточно маленькие, чтобы быстро двигаться без проблем с коммуникацией, которые часто возникают в крупных командах.

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 иных 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 состоят в том, что связи внутри системы становятся слабее, а значит, отладка системы в случае отказа намного усложняется. 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.

Кроме того, в самих командах может наблюдаться стремление довести паттерн микросервисов до крайности, в результате чего получается слишком много очень маленьких микросервисов. Такое чрезмерное дробление не только увеличивает общую сложность, но и может существенно ухудшить эффективность с точки зрения повышения накладных расходов на сетевые взаимодействия (поскольку микросервисы общаются друг с другом по сети), а также с точки зрения расходования ресурсов (поскольку каждый экземпляр микросервиса имеет фиксированный объем накладных расходов на потребление таких ресурсов, как процессорное время и память). Как правило, если количество микросервисов, разрабатываемых командой, превышает количество инженеров в ней, это означает, что у вас либо слишком маленькая команда, либо вы переусердствовали с микросервисным подходом.

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, так как 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 взаимоисключающими. Любая реальная система строится на основе набора паттернов, взаимодействующих в рамках одного высокоуровневого приложения.

ГЛАВА 6

Реплицированные сервисы с распределением нагрузки

Реплицированный сервис с распределением нагрузки — простейший паттерн распределенных вычислений, известный многим. В рамках такого сервиса все серверы идентичны друг другу и одинаково обрабатывают входящий трафик от любого клиента. 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 приложениях без состояния (stateless-приложениях) отдельные запросы могут направляться разным экземплярам сервиса (рис. 6.1). Примеры stateless-сервисов включают как сервисы доставки статического контента, так и сложные промежуточные системы, принимающие и агрегирующие запросы от внутренних сервисов.

Системы без состояния 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 три девятки* в день есть 1,4 минуты на простой (24 × 60 × 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 того, что вы обновляете программное обеспечение раз в день. Если ваша команда серьезно вступила на путь непрерывной доставки 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. И все это при условии, что ваш код никогда не потерпит сбой из-за ошибок, что практически невозможно в реальной жизни.

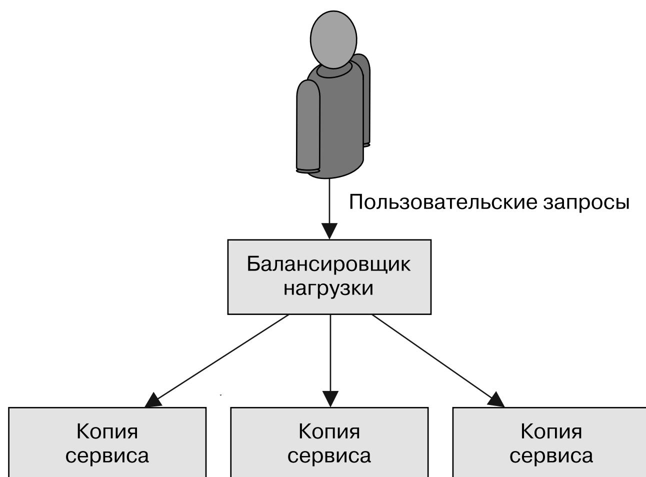


Рис. 6.1. Простой реплицированный stateless-сервис

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. 6.2). Это происходит благодаря использованию паттерна Load-balanced Replicated Serving (обслуживание с репликацией и балансировкой нагрузки).

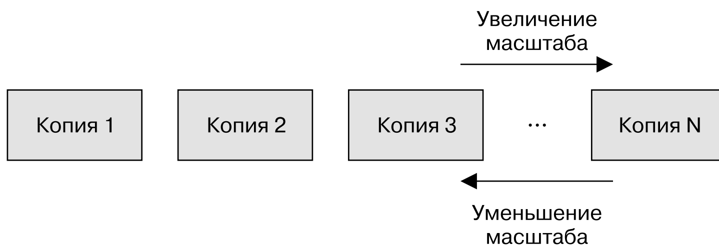


Рис. 6.2. Горизонтальное масштабирование реплицированного stateless-приложения

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 Node.js-приложение, которое выводит толкования слов из словаря.

Сервис можно опробовать в контейнере, выполнив такую 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, используем объект `Deployment`:

```
apiVersion: extensions/v1
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 в кэш. Это также может быть обусловлено долгосрочной природой взаимодействия пользователя с системой, когда часть состояния сохраняется 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. 6.3).

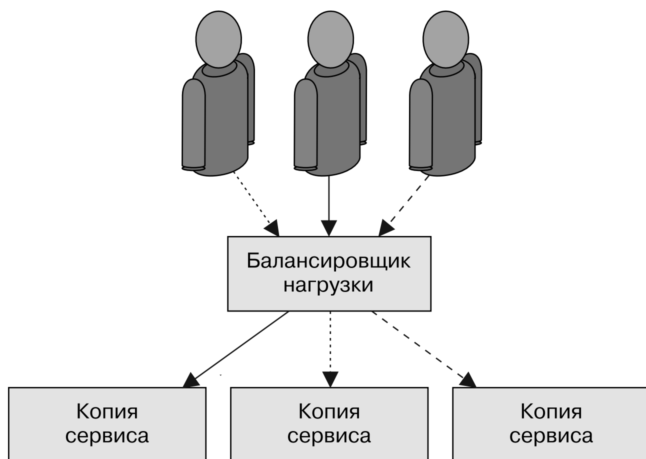
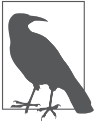


Рис. 6.3. Сервис с закреплением сессий, в котором все запросы конкретного пользователя адресуются одному и тому же экземпляру сервиса

Вообще говоря, закрепление сессий реализуется путем хеширования IP-адресов клиента и сервера и использования этого хеша для выбора экземпляра сервиса, который будет обслуживать запрос. Пока IP-адреса клиента и сервера остаются неизменными, все запросы от одного и того же клиента будут направляться одному и тому же экземпляру сервиса.



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).

Закрепление сессий часто реализуется на базе *согласованной хеш-функции*. Преимущества согласованной хеш-функции становятся очевидны при масштабировании сервиса в большую или меньшую сторону. Конечно, если количество копий сервиса меняется, то может поменяться и соответствие между конкретным пользователем и конкретным экземпляром сервиса. Согласованные 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 кэширующий слой

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 кэширующий слой. 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. 6.4).

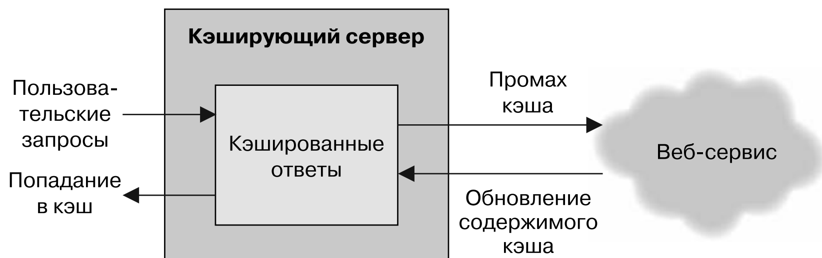


Рис. 6.4. Работа кэширующего сервера

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. 6.5).

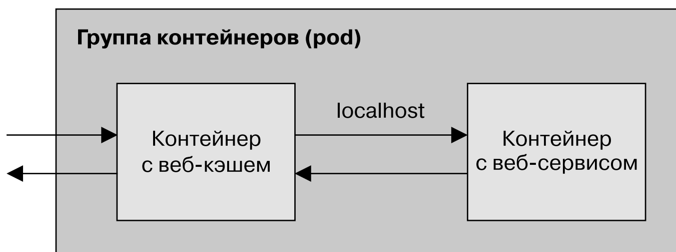


Рис. 6.5. Добавление кэширующего веб-сервера в виде контейнера-прицепа

Такой подход при всей простоте имеет недостатки. В частности, вам придется масштабировать кэш одновременно с веб-сервером. 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 страница будет записана десять раз, что уменьшит общее количество разных страниц, одновременно находящихся в кэше. Это снижает *коэффициент попадания в кэш* — долю запросов, обслуживаемых из кэша, что, в свою очередь, уменьшает полезность кэша. Поэтому часто желательно иметь лишь несколько крупных кэшей и как можно больше небольших экземпляров веб-серверов. Многие языки, например Node.js, 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 кэширующий слой как второй реплицированный stateless-сервис, находящийся над веб-сервисом (рис. 6.6).

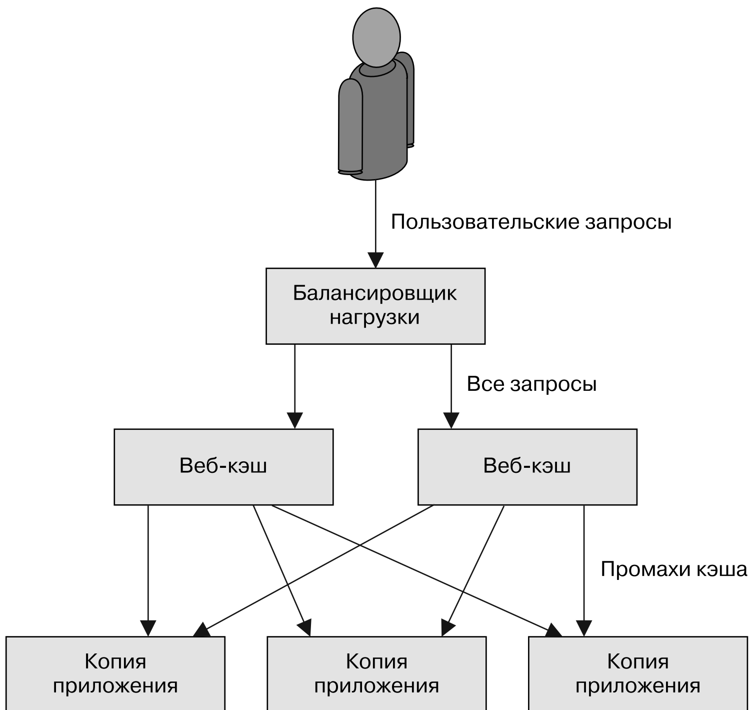
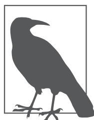


Рис. 6.6. Добавление кэширующего слоя к реплицированному сервису



Если не соблюдать осторожность, кэширование может нарушить работу механизма 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. 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.

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. 6.7.

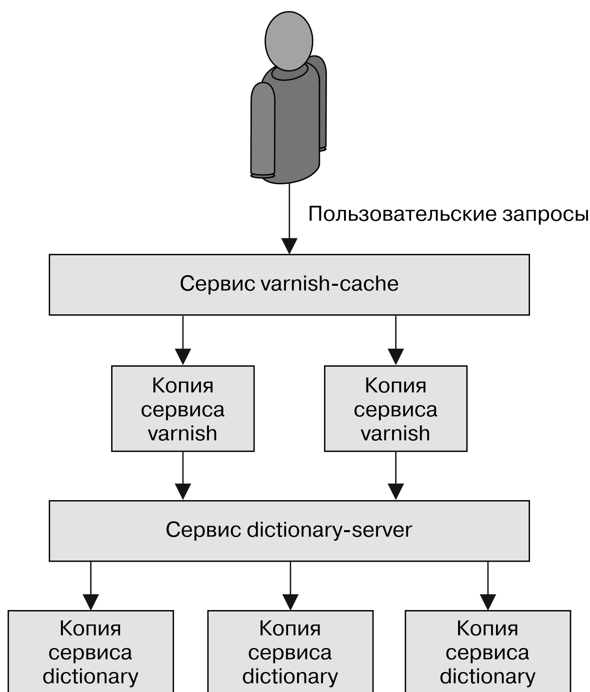


Рис. 6.7. Добавление кэширующей прослойки к серверу-словарю

Начнем создание кэширующего слоя с настройки кэширующего сервера Varnish:

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

Создадим объект `ConfigMap`, содержащий указанную конфигурацию:

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

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

```
apiVersion: extensions/v1
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
          # Объем выделяемой здесь памяти должен соответствовать
          # объему 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
```

Это можно сделать с помощью такой команды:

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

Расширение возможностей кэширующего слоя

Теперь, когда мы развернули кэширующий слой 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.

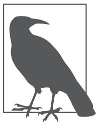
Ограничение частоты запросов и защита от атак типа «отказ в обслуживании» (DoS)

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 клиента, либо из-за

того, что инженер, ответственный за доступность сайта, случайно запустил нагрузочные тесты на рабочем сервере. Следовательно, имеет смысл добавить в кэширующий слой 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.

Развертывая API, целесообразно предусмотреть 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, маскирующихся под нескольких пользователей, чтобы успешно реализовать атаку.

Когда количество запросов от одного пользователя достигнет определенного лимита, сервер вернет ему ошибку с кодом 429, означающую превышение количества максимально допустимых запросов. Однако многие пользователи хотели бы знать, сколько еще запросов они могут сделать, 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`.



Несмотря на наличие в Varnish некоторой базовой защиты от DDoS-атак, в реальном мире злоумышленники часто объединяются в сложные организации с большим количеством ресурсов, которые можно задействовать в постоянно меняющихся атаках. Если вы планируете развернуть крупномасштабную или критически важную инфраструктуру, то, скорее всего, также захотите добавить в свое приложение поддержку — облачную защиту от атак DDoS, предлагаемую поставщиками облачных услуг. Облачные системы DDoS могут выдерживать значительные нагрузки и к тому же постоянно развиваются, чтобы соответствовать меняющемуся ландшафту субъектов угроз.

SSL-мост

Вдобавок к кэшированию для достижения еще большей производительности пограничный слой приложения также может выполнять функции SSL-моста. Даже если вы планируете использовать SSL для взаимодействия между внутренними слоями приложения, 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, веб-кэш 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. 6.8).

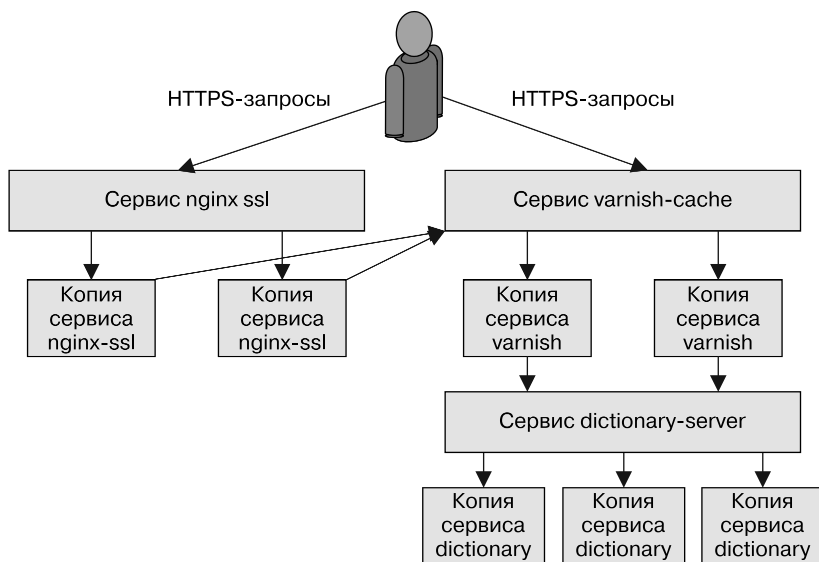
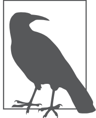


Рис. 6.8. Пример реплицированного stateless-сервиса

Практикум. Развертывание nginx и SSL-моста

Следующие инструкции описывают, как добавить SSL-мост на основе `nginx` к уже развернутому реплицированному сервису и кэшу.



Они предполагают наличие у вас сертификата. Если вам нужно получить сертификат, то проще всего сделать это с помощью инструментов Let's Encrypt (https://letsencrypt.org/). Для создания сертификатов также можно воспользоваться инструментом `openssl`. Данные инструкции также предполагают, что файл сертификата носит имя `server.crt`, а файл закрытого 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 можно создать слой реплицированных stateless-серверов nginx:

```
apiVersion: extensions/v1
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 защиты соединений с клиентами. Полный паттерн реплицированного stateless-сервиса представлен на рис. 6.8.

Его можно развернуть в Kubernetes с помощью трех объектов `Deployment` и трех объектов `Service` с балансировщиками нагрузки. Полные исходные тексты примеров можно найти по адресу https://github.com/brendandburns/designing-distributed-systems.

ГЛАВА 7

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)* анализирует каждый запрос и направляет его соответствующему шарду (или шардам) для обработки. Разница между реплицированными и шардированными сервисами показана на рис. 7.1.

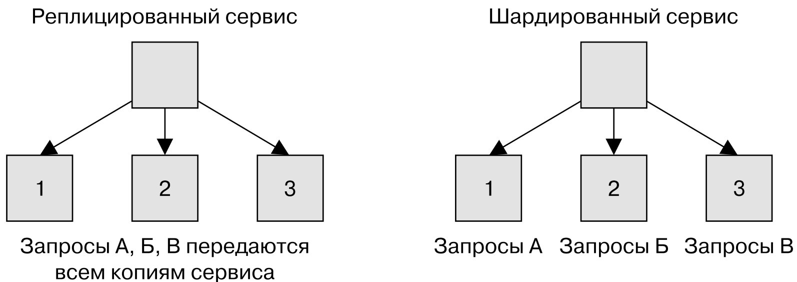


Рис. 7.1. Реплицированные и шардированные сервисы

Репликация сервиса обычно используется для построения stateless-сервисов, а шардирование — для сервисов с состоянием (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.

Однако шардировать можно не только сервисы с состоянием. Иногда шардинг может пригодиться в сервисах без состояния для их изоляции. Сбои в распределенных системах могут возникать из-за входящих запросов к реплицированной системе (такие запросы иногда называют «ядовитыми» или «отравляющими») — если «ядовитых» запросов окажется достаточно много, то они смогут вывести из строя все реплики. В таких системах шардинг может помочь провести границы изоляции, защищающие от «ядовитых» запросов. Даже притом, что «ядовитые» запросы смогут вывести из строя весь шард, это лишь часть запросов, которые обрабатывает система.

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. 7.2.

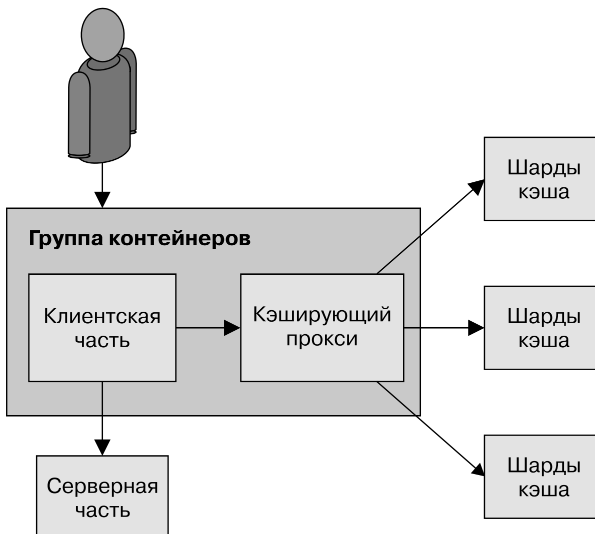


Рис. 7.2. Шардированный кэш

В главе 4 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 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, 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 кэш, разбитый на десять шардов, 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 эффективнее, поскольку каждый элемент данных попадает только в один кэш. Чем больше данных хранится в кэше, тем больше запросов будет обслуживаться из кэша и тем быстрее сможет работать вся система.

Роль кэша в производительности системы

В главе 6 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 промахи кэша 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 в кэш*. Коэффициент попадания — доля запросов, ответ на которые содержится в кэше. В конечном счете коэффициент попадания характеризует общую максимальную нагрузку на распределенную систему и влияет на производительность и мощность системы в целом.

Представьте, что уровень обработки запросов в вашем приложении способен обрабатывать до 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, уменьшится 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 сложнее в реализации и развертывании, но имеет определенные преимущества перед просто шардированным сервисом. Что особенно 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 4 мы рассмотрели процесс развертывания шардированного сервиса Redis. Развертывание шардированного сервиса memcache происходит подобным образом.

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

```
apiVersion: apps/v1
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` и разверните его 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-объект `Service`, 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 его командой `kubectl create -f memcached-service.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://oreil.ly/IFxFA).

```
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 этой конфигурации видно, что запросы к memcache обслуживаются по адресу `localhost:11211`, 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 можно развернуть пример реализации паттерна Ambassador. Определим 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 код в файл с именем `memcached-ambassador-pod.yaml`, затем разверните его 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 через все интерфейсы, а не только через петлевой:

```
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/v1
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
```

И развернуть его 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. Теперь посмотрим, как трафик переадресуется тому или иному шарду. Рассмотрим шардированный сервис

с десятью независимыми шардами. Дан пользовательский запрос *Req*. Как выяснить, какой из шардов *Shard* с номерами от 0 до 9 должен обработать запрос? Задача выбора шарда возлагается на шардирующую функцию. 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. Детерминированность важна, так как обеспечивает попадание определенного запроса *Req* в 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 хеш-функцию из стандартной библиотеки 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 отметки времени в запросах не совпадают. Но в большинстве случаев ответ на HTTP-запрос не зависит ни от адреса клиента, ни от отметки времени запроса. Следовательно, предпочтительнее 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, зависящей только от HTTP-пути в запросе, 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.

Выбирать номер шарда в зависимости от IP-адреса и HTTP-пути одновременно — 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 как часть 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.

Согласованные 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, не на те, что назначались предыдущей функцией. Это существенно увеличит процент промахов кэша в шардированном кэше, пока шарды не заполнятся ответами на вновь сопоставленные с ними запросы. Применение новой функции шардирования к шардированному кэшу в худшем случае приведет к промахам кэша в 100 % случаев.

Чтобы решить подобную проблему, многие шардирующие функции прибегают к использованию *согласованных 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 согласованной 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 согласованного 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 согласованную хеш-функцию.

Шардирование реплицированных сервисов

В большей части примеров этой главы 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 можно шардировать 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. 7.3. 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.

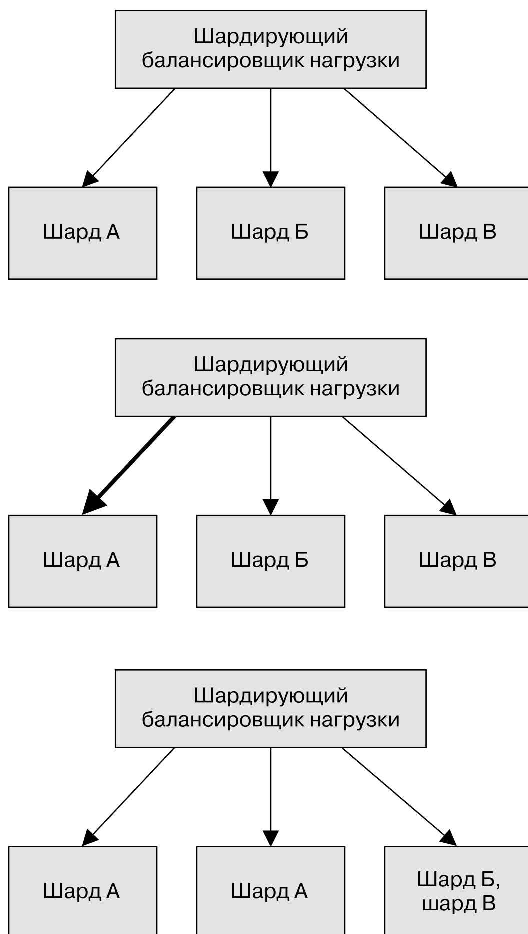


Рис. 7.3. Пример системы с «горячим» шардированием. Сначала шарды нагружены равномерно, но когда нагрузка на шард А увеличивается, он реплицируется на две машины, а шарды Б и В попадают на одну

Резюме

В этой главе была представлена концепция шардинга — направления определенных запросов определенным группам машин вместо балансировки нагрузки между репликами сервиса. Вы увидели, что шардинг может значительно повысить эффективность и производительность системы кэширования, подключенной к обслуживающему API. Однако шардинг может применяться не только для кэширования. Такие системы, как крупномасштабные игровые серверы с открытым миром и базы данных, можно масштабировать за пределы одной машины, распределяя данные между несколькими шардированными серверами. Шардинг также помогает изолировать сервисы и ограничить радиус поражения в случае атак «ядовитыми» запросами.

Настройка шардированной системы требует внимательного подхода к выбору ключа шардинга и хеш-функции. Определение правильного способа шардирования данных часто играет важную роль для масштабирования системы. Понимание деталей шардирования имеет решающее значение для построения эффективных, крупных и сложных распределенных систем.

ГЛАВА 8

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, Scatter/Gather — древовидный паттерн, в котором корневой узел распределяет запросы, а листовые 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 корневому узлу. Корневой узел затем собирает частичные ответы в один общий ответ, который и возвращается клиенту. Паттерн Scatter/Gather схематически изображен на рис. 8.1.

Он весьма полезен, если обработка запроса заключается в выполнении множества независимых операций. В таких задачах довольно легко можно распределить требуемые вычисления по нескольким независимым процессам.

Если в предыдущих примерах кэширования мы производили шардинг данных, чтобы максимально эффективно использовать память, то в примерах реализации паттерна Scatter/Gather мы будем шардировать вычисления, чтобы уменьшить задержку за счет распараллеливания обработки запросов. Хотя иногда, как мы увидим, шардированию подвергаются и данные.

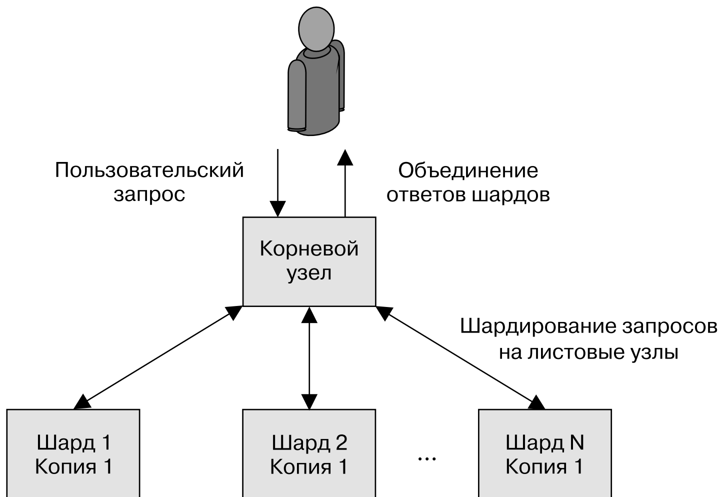


Рис. 8.1. Паттерн Scatter/Gather

Scatter/Gather с распределением нагрузки корневым узлом

В простейшем варианте паттерна Scatter/Gather все листовые 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.

Достичь 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 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 читать 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 поиска совпадений во всех документах можно воспользоваться простым поиском в ассоциативном массиве. Правда, при этом теряется одна важная возможность. Напомню, что мы ищем все документы, включающие оба слова, «кот» *и* «собака», тогда как индекс содержит только единичные слова, а не наборы слов. К счастью, множество документов, включающих оба слова, представляет собой пересечение множеств документов, содержащих отдельные слова.

Мы могли бы реализовать такой поиск на одной машине. На первом шаге выполнить поиск в индексе документов, содержащих слово

«кот». На втором шаге — документов, содержащих слово «собака». И на последнем шаге выбрать документы, присутствующие в обоих списках. Как видите, поиск по слову «кот» и поиск по слову «собака» не зависят друг от друга и могут выполняться параллельно и даже на разных машинах. Однако вычисление пересечения требует обоих списков документов и поэтому зависит от результатов первых двух шагов.

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 передает для обработки 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. 8.2: листовой 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}`.

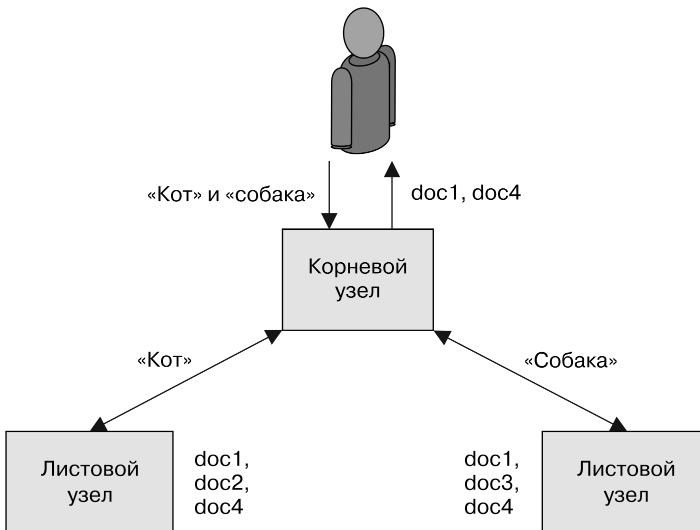


Рис. 8.2. Пример поисковой системы Scatter/Gather с шардированием по термам поискового запроса

Scatter/Gather с шардированием листовых 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 выбора узла, которому будет отправлен запрос, использовался некоторый его фрагмент. Этот узел впоследствии полностью обрабатывал запрос и выдавал ответ пользователю. В случае шардирования Scatter/Gather, напротив, запрос отправляется всем листовым узлам (шардам) в системе. Каждый листовой узел обрабатывает запрос, используя данные, содержащиеся в своем шарде. Частичный ответ возвращается корневому узлу, а тот 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. 8.3.

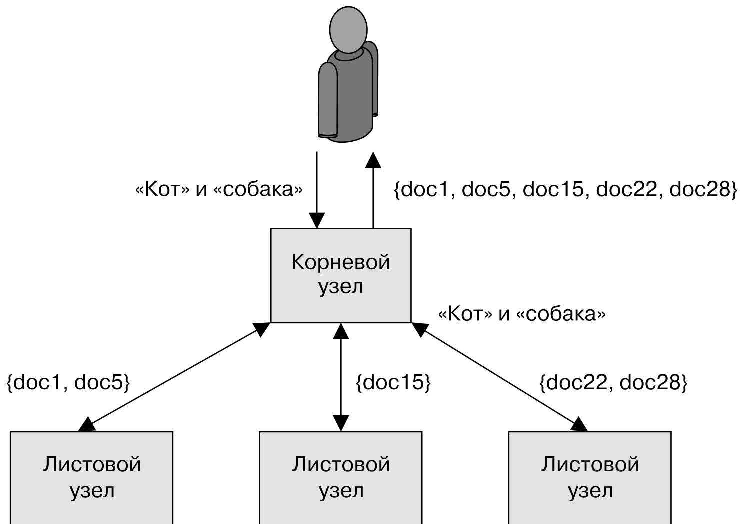


Рис. 8.3. Выполнение запроса с конъюнкцией в поисковой системе типа Scatter/Gather

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 листовом узле дерева недостаточно 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 листовым узлам Scatter/Gather-дерева. Каждый листовой узел возвращает набор известных ему документов, содержащих слова «кот» и «собака». Ранее корневой узел отвечал за вычисления пересечения 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.

Первый листовой узел на рис. 8.3 обслуживает документы 1–10 и возвращает `{doc1, doc5}`. Второй листовой узел обслуживает документы 11–20 и возвращает `{doc15}`. Третий листовой узел обслуживает документы 21–30 и возвращает `{doc22, doc28}`. Корневой узел объединяет ответы и возвращает множество `{doc1, doc5, doc15, doc22, doc28}`.

Выбор подходящего количества листовых 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. Однако увеличение степени распараллеливания влечет дополнительные расходы, поэтому для достижения максимальной производительности в распределенной системе чрезвычайно важно правильно выбрать количество листовых узлов.

Чтобы понять, почему так происходит, следует принять во внимание два аспекта. 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 постоянны и сравнительно невелики по отношению к 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 растет с увеличением количества листовых узлов. Каждый конечный узел, который вы добавляете для увеличения степени параллелизма, увеличивает общие накладные расходы. Кроме того, увеличение степени параллелизма сокращает время, затрачиваемое на выполнение «полезных» вычислений каждым листовым узлом, и в какой-то момент время, затрачиваемое на накладные расходы, может значительно превысить время, затрачиваемое на выполнение «полезных» вычислений. Когда это происходит, становится невозможным повышение

производительности за счет добавления дополнительных листовых узлов. А это значит, что рост производительности за счет распараллеливания носит асимптотический характер. Вы не можете безгранично добавлять узлы и ожидать повышения производительности.

Помимо того что добавление листовых 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* листовых 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 системах, построенных по принципу Scatter/Gather, дела обстоят несколько иначе. Поскольку время обработки запроса определяется самым медленным узлом, нам приходится рассматривать не один запрос, а все запросы, распределенные между листовыми узлами системы.

Посмотрим, что произойдет, если распределить запросы на пять листовых узлов. Вероятность того, что обработка запроса одним из 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 листовых узлов, а вероятность отказа любого из них 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 листовыми 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 счете 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, листовые узлы будут реализованы не в виде отдельных шардов, а в виде реплицированных сервисов. Схема реплицированного шардированного паттерна Scatter/Gather изображена на рис. 8.4.

Нагрузка на листовой 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 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.

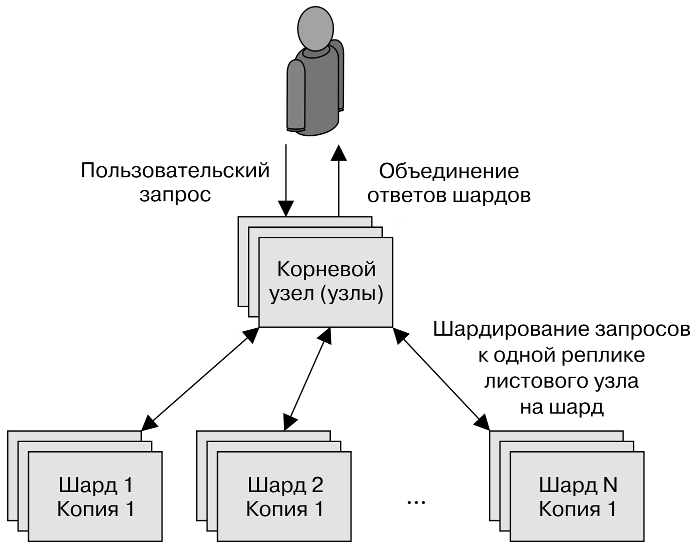


Рис. 8.4. Scatter/Gather-система с шардированием и репликацией

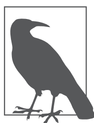
Резюме

В этой главе описывается паттерн Scatter/Gather распределенных систем для повышения скорости вычислений и уменьшения задержек при обработке любого конкретного запроса. В отличие от предыдущих паттернов обслуживания, сосредоточенных на масштабировании нагрузки или данных, паттерн Scatter/Gather масштабирует вычисления, повышая отзывчивость и сокращая воспринимаемое время обработки запроса. Вы также увидели, что Scatter/Gather позволяет объединять шардинг вычислений с шардингом данных, а также добавлять репликацию для повышения надежности и избыточности. Любая крупномасштабная распределенная система часто реализуется с применением комбинации нескольких паттернов обслуживания.

ГЛАВА 9

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. Однако 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).



Часто подход FaaS называют *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.

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.

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 задач порождает сложные, хрупкие архитектуры. Поскольку подход FaaS еще относительно 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 кодом и работающим сервисом. Поскольку, за исключением исходного кода, нет необходимости создавать и разворачивать другие артефакты, FaaS упрощает переход от исходного кода, открытого в текстовом редакторе на ноутбуке или в браузере, к работающему 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 подразделе 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 исходном коде. 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 развития 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 не годится для приложений, требующих длительной фоновой обработки данных. В качестве примера можно привести такие задачи, как перекодирование видео, сжатие файлов журналов и другие продолжительные 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 обработки 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, особенно если учесть, что стоимость облачных виртуальных машин сокращается по мере добавления ядер, а также за счет резервирования ресурсов и скидок за длительное пользование, тогда как 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 описывается 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. 9.1.

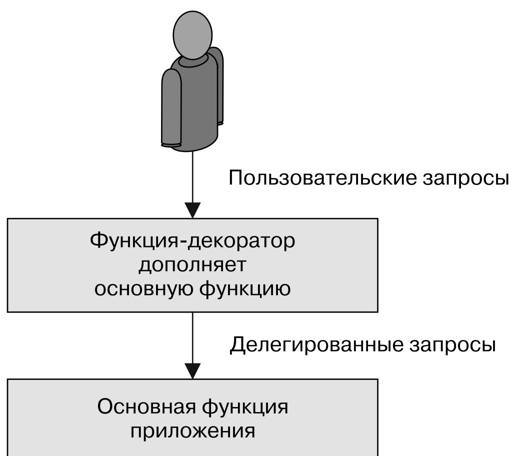


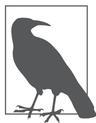
Рис. 9.1. Применение паттерна Decorator при проектировании HTTP API

К слову, в языках программирования существует несколько аналогий данного паттерна. В частности, в Python есть *декораторы* функций, которые функционально похожи на декораторы запросов 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, вам может понадобиться, чтобы по умолчанию поле хранило значение `true`, но это трудно реализовать с помощью классического JSON, потому что 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 решения 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, вы, возможно, зададитесь вопросом, 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://oreil.ly/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://oreil.ly/kubelessreleases). 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 использовать в инструменте 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 равным '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 с именем `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
```

Для того 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, давая возможность разработчику сосредоточиться на шагах, необходимых для обработки только одного конкретного типа событий.

Конкретный пример интеграции событийно-ориентированного компонента в существующий сервис — 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 требует, чтобы пользователь что-то знал (например, пароль) и что-то имел (например, телефон) для входа в систему. Двухфакторная аутентификация намного лучше просто пароля, поскольку злоумышленнику для получения доступа придется украсть и ваш пароль, и ваш телефон.

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"}
```

Эту функцию FaaS можно зарегистрировать с помощью `kubeless`:

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

Экземпляр этой функции можно асинхронно вызывать из клиентского кода на JavaScript после ввода пользователем правильного пароля. Веб-интерфейс может немедленно отобразить страницу для ввода кода, а пользователь, как только получит код, сможет 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 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.

Так в чем же разница между таким конвейером и микросервисной архитектурой? Важных различий два. Первое и самое главное различие между сервисами-функциями и постоянно работающими сервисами состоит в том, что событийные конвейеры, по сути, управляются

событиями. Микросервисная архитектура, 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 управления версиями. 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 размещение логики создания новых учетных записей в одном монолитном сервисе. 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)
```

Факторизованный FaaS-сервис выглядит 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 его применение упрощает управление 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.

Резюме

Функция как сервис (function-as-a-service, FaaS) — мощный инструмент для размещения простых масштабируемых приложений. Как и для любого другого инструмента, ключом к его эффективному использованию является понимание сферы его применения. Простота FaaS-функций влечет за собой ограничения на подходы к разработке приложений или библиотек, используемых вашими приложениями. Однако FaaS также может радикально сократить время выхода на рынок. С накоплением опыта работы с распределенными системами вы начнете замечать, что идеальная конструкция часто представляет собой комбинацию разных компонентов. Объединение FaaS с более гибкой инфраструктурой или примитивами IaaS (infrastructure-as-a-service — «инфраструктура как сервис»), такими как ядро Kubernetes API, часто позволяет взять все самое лучшее из обоих миров и получить оптимальную систему с оптимальной конструкцией, объединив мощь и гибкость с простотой и автоматизацией.

ГЛАВА 10

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.

Возможно, вы уже умеете управлять параллелизмом в контексте одного приложения на одном сервере с помощью таких примитивов, как *блокировки* или *мьютексы*, поддерживаемых современными языками программирования. Блокировка устанавливает владение в контексте одного приложения, работающего на одной машине, потому что использует хранилище в памяти этой машины и примитивы в процессоре и операционной системе для обеспечения исключительного доступа. Но даже в контексте одной машины реализация параллельных вычислений — непростая задача, а ошибки в реализации — одни из самых сложных, с которыми можно столкнуться.

К сожалению, 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 управления блокировками. Разработка распределенных блокировок сопряжена с дополнительными сложностями, потому что вы уже не можете положиться на общую память или процессоры, совместно используемые разными контейнерами в распределенной системе.

Общая схема распределенного владения приводится на рис. 10.1. 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.

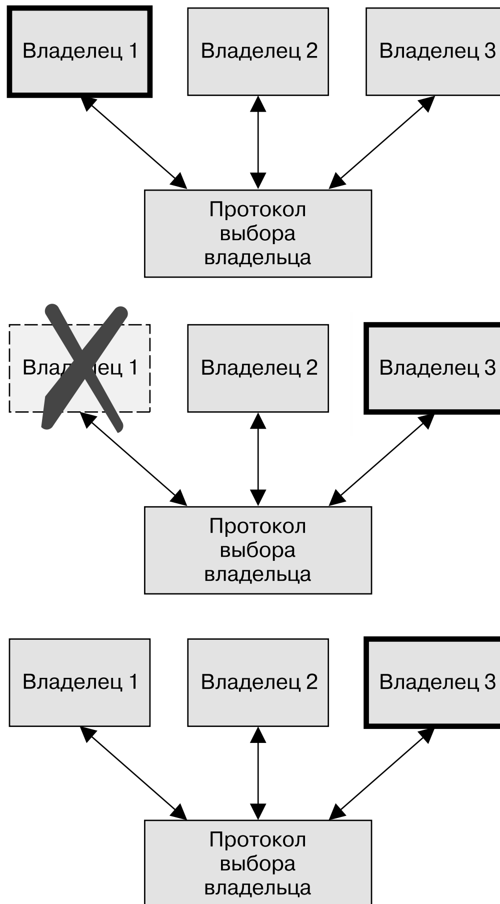


Рис. 10.1. Протокол выбора владельца в действии: сначала владельцем является первый экземпляр сервиса, а в случае его отказа полномочия берет на себя третий экземпляр

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. Рассмотрим этот момент подробнее.

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 некоторое время. Старая и новая версии синглтона не могут работать одновременно, поэтому на время обновления придется остановить 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.

Использование сервиса-одиночки имеет существенный недостаток — в период, когда лидер недоступен, все приложение не сможет обрабатывать запросы. Вы можете достичь хорошего времени безотказной работы, исчисляемого сутками, но в течение тех секунд или минут, когда сервис может быть недоступен, недоступным будет и все приложение. Для интерактивных систем, таких как игровые серверы, розничная или банковские системы, полный отказ даже на короткий период времени может вызвать значительный всплеск недовольства клиентов.

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 выбор начального владельца, а в случае отказа текущего — очередного владельца, называется *выбором владельца*.

Есть два способа реализовать выбор владельца. Например, можно 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:

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

func compareAndSwap(key, nextValue, currentValue string) (bool, error)
{
  lock.Lock()
  defer lock.Unlock()
  _, containsKey := store[key]
  if !containsKey {
    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
}
```

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://etcd.io/) — распределенный сервис блокировок. Первоначально был разработан в рамках проекта CoreOS, но затем передан под эгиду фонда Cloud Native Computing Foundation (CNCF) (https://www.cncf.io) и в настоящее время развивается сообществом. Он довольно устойчив и хорошо себя зарекомендовал в различных проектах, в том числе в Kubernetes.

Проще всего запустить `etcd` локально. Проект предоставляет двоичные файлы для Linux, MacOS X и Windows. Найти эти файлы можно на GitHub (https://oreil.ly/WqiLp). На момент написания этих слов последней была версия 3.5.16, но вам может оказаться доступной более свежая версия.

После загрузки архива его нужно распаковать, в результате чего вы получите три исполняемых двоичных файла, `etcd`, `etcdctl` и `etcdutl`, а также файлы с документацией. Для опробования этого упражнения достаточно запустить `etcd` локально, выполнив команду `etcd`. Она запустит `etcd` на локальном хосте (127.0.0.1).

Очевидно, что запуск `etcd` локально не самое надежное решение. При желании `etcd` можно запустить внутри кластера Kubernetes с помощью менеджера пакетов с открытым исходным кодом Helm. Если у вас еще не установлен Helm, то вы можете установить его, получив дистрибутив на сайте Helm (https://helm.sh).

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

```
# Инициализация helm
helm init

# Установка оператора etcd
helm install my-release oci://registry-1.docker.io/bitnamicharts/etcd
```

После установки оператора `etcd` можно выполнить команду `kubectl get pods`, чтобы увидеть реплики в кластере `etcd`.

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 в 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:

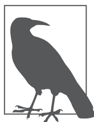
```
func (Lock l) lock() {
  while (!l.simpleLock()) {
    waitForChanges(l.lockName)
  }
}
```

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. Если этого не сделать, то есть риск, что программа продолжит вычисления после потери блокировки.

Добавив к блокировкам TTL, мы внесли 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. Мы можем проверять версию ресурса в любой последующей операции записи, чтобы гарантировать упорядоченность. Наша функция установки блокировки сохраняет версию ресурса и использует ее в `compareAndSwap`, чтобы проверить совпадение не только значения, но и версии ресурса при снятии блокировки. В результате наша простая функция `simpleLock` изменится, как показано ниже:

```
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. Блокировку не получится снять случайно, если кто-то другой получил ее после истечения TTL, потому что в этом случае новый владелец блокировки должен записать значение в систему хранения, что приведет к обновлению версии ресурса и наша попытка разблокировки с использованием старой версии ресурса потерпит неудачу.

Практикум. Реализация блокировок в etcd

При реализации блокировок в `etcd` ключи можно использовать в качестве 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, но в реальных условиях вам следует воспользоваться языком программирования. Клиенты `etcd` есть для всех популярных языков программирования.

Начнем с создания блокировки с именем `my-lock`:

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

Эта команда 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"
```

и устанавливает 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. Но такое решение имеет существенный недостаток — 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=<время в секундах>`, 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"
```

Она создаст арендованную блокировку, которая будет удерживаться 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 тоже 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. 10.2. Блокировкой владеет `shard2`, и, когда рабочему узлу отправляется запрос, тот уточняет на сервере блокировок, действительно ли `shard2` является текущим владельцем.

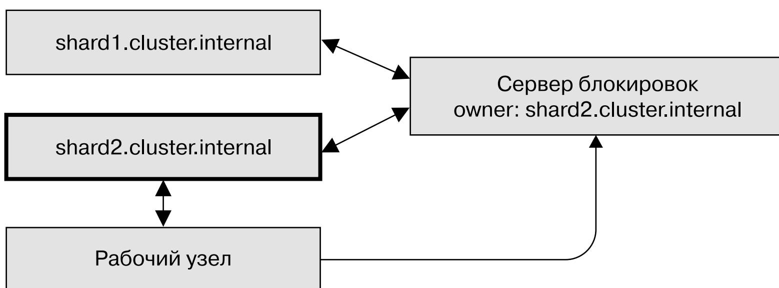


Рис. 10.2. Рабочий узел проверяет, что отправитель запроса действительно является текущим владельцем шарда

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 `R1` 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`.
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, но беда в том, что `R2` уже исполнен.

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.

Учитывая все вышесказанное, вам придется при проектировании распределенной системы владения выбирать между семантиками обработки «не более одного раза» (что может потребовать принять возможность потери доступности на некоторый период времени) и «не менее одного раза» (что обеспечит более высокую доступность,

но иногда будет приводить к двойной обработке запросов). В зависимости от типа создаваемой системы, двойная обработка может просто тратить некоторые ресурсы (например, время вычислений на GPU). Если такое происходит нечасто, то это, вероятно, вполне приемлемо. Однако в некоторых системах, особенно выполняющих финансовые трансакции (например, переводящие деньги между счетами), предпочтительнее потерять доступность, но сохранить согласованность. Понимание этих компромиссов является важнейшей частью проектирования распределенных систем.

Резюме

В этой главе мы рассмотрели один из самых сложных аспектов проектирования распределенной системы — распределенное владение и блокировки. Параллельная обработка является ключевой особенностью надежных масштабируемых систем, но параллелизм также создает проблемы в системах, где важно владение определенными ресурсами или единицами работы. Распределенное владение или блокировки позволяют частям распределенной системы поддерживать эксклюзивный доступ к определенной части хранилища данных или вычислительных операций. Реализация блокировок подразумевает взаимодействие с распределенным хранилищем на основе консенсуса, таким как `etcd` или ZooKeeper, с очень специфическими паттернами программирования.

Во многих языках распределенные блокировки уже реализованы. Самостоятельная разработка такого кода является чрезвычайно сложной задачей, поэтому если для вашего языка программирования уже есть готовая библиотека, то лучше положиться на готовую реализацию промышленного уровня. Наконец, некоторые простые системы не нуждаются в блокировках и могут обойтись использованием объектов-одиночек и внутрипроцессного параллелизма. Такие системы жертвуют доступностью ради простоты.

Часть IV

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: генерация сводки по данным пользовательской телеметрии, анализ данных продаж для ежедневной или еженедельной отчетности, преобразование формата видеофайлов. 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 главах этой части.

ГЛАВА 11

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 обрабатываться 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. 11.1.

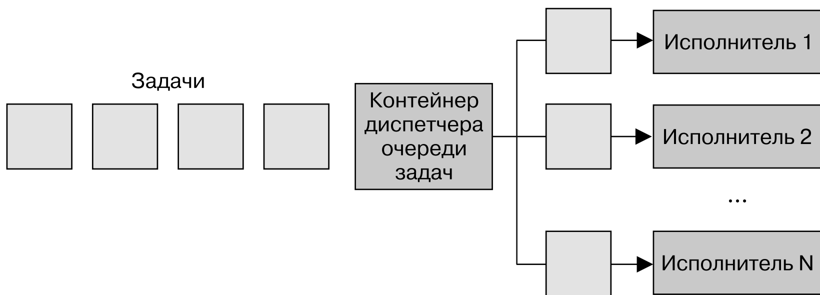


Рис. 11.1. Обобщенная очередь задач

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 рода выполняемой работы. Во многих случаях то же касается и доставки самих задач.

Проиллюстрирую данное утверждение с помощью очереди задач, изображенной на рис. 11.1. Посмотрев на нее еще раз, определите, какие ее функции могут быть предоставлены совместно используемым набором контейнеров. Становится очевидным, что большая часть реализации контейнеризированной очереди задач может использоваться широким спектром пользователей, как показано на рис. 11.2.

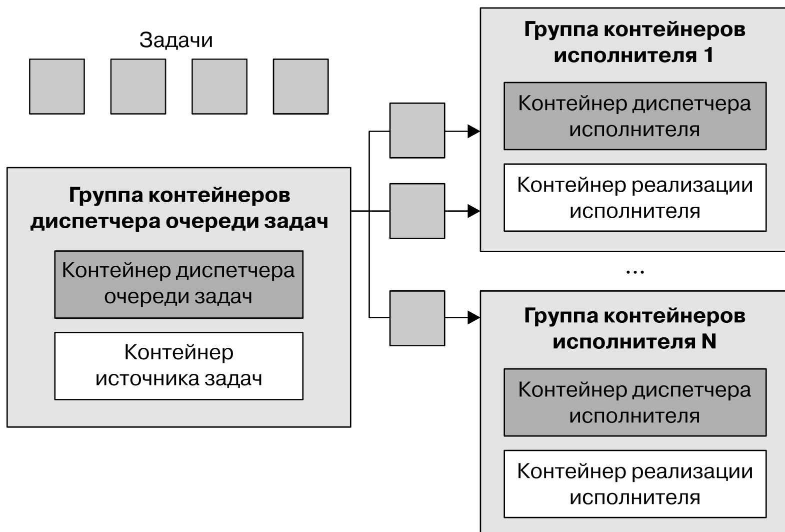


Рис. 11.2. Та же очередь, что и на рис. 11.1, но на этот раз с использованием многоразовых контейнеров. Многоразовые контейнеры показаны белым цветом, а пользовательские — более темным серым цветом

Построение очереди задач на основе контейнеров требует согласования интерфейсов между библиотечными контейнерами и контейнерами с пользовательской логикой. В рамках контейнеризированной 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 выступает послом, транслирующим запросы контейнера диспетчера очереди конкретным исполнителям задач. Данная группа контейнеров изображена на рис. 11.3.

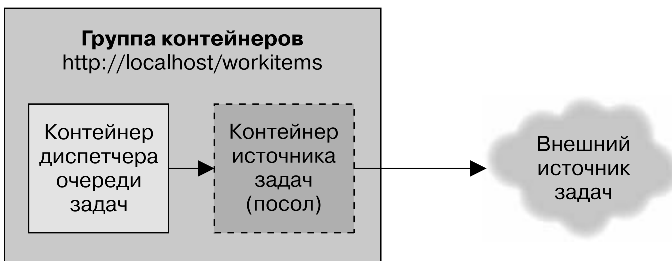


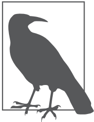
Рис. 11.3. Группа контейнеров, реализующая очередь задач

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 является не только самым простым в реализации, но и 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 бóльшую часть реализации на контейнер-посол. Помните, однако, что наша цель — сосредоточить как можно бóльшую 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` интерфейсу контейнера-исполнителя для обработки.

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 `WORK_ITEM_FILE`, ссылающуюся 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. 11.4).

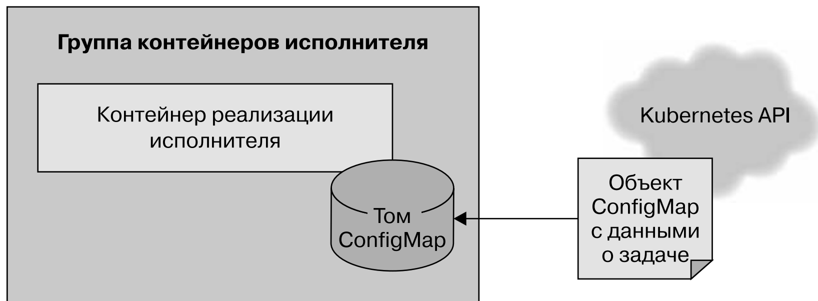


Рис. 11.4. API контейнера-исполнителя

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, бóльшая 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, бóльшая 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 выполнение задачи.

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, поскольку оркестратор берет на себя ответственность за надежное исполнение задач.

Кроме того, Kubernetes позволяет аннотировать задачи, что дает нам возможность пометить каждый объект `Job` названием обрабатываемой задачи. 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 построение инфраструктуры очереди задач.

Подробный алгоритм работы контейнера — диспетчера очереди задач выглядит следующим образом.

```
Повторять бесконечно.
    Получить список задач посредством интерфейса контейнера —
    источника задач.
    Получить список заданий, обслуживающих данную очередь задач.
    Выделить на основе этих списков перечень необработанных задач.
    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
```

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

Для создания контейнера-исполнителя выберите базовый образ по своему вкусу и установите в него исполняемый файл `ffmpeg`. Можно также воспользоваться проектом на GitHub с открытым исходным кодом, содержащим файлы Dockerfile для многих популярных операционных систем (https://oreil.ly/NJj2S), и на его основе создать свой образ контейнера.

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 не страшно, 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 одна 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 отставать. 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 % межзадачного интервала. Иногда эффективнее создать свой механизм масштабирования, но чаще предпочтительнее использовать готовое решение с открытым исходным кодом. Одно из таких решений предлагает проект Kubernetes Event-Driven Autoscaling (KEDA) (https://keda.sh). KEDA предоставляет множество различных реализаций масштабирования на основе событий и поддерживает самые разные источники событий, начиная от файлов на диске и заканчивая событиями в системах публикации/подписки и многими другими.

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 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, например, для размытия на фото чего-нибудь еще, скажем машин.

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. 11.5.

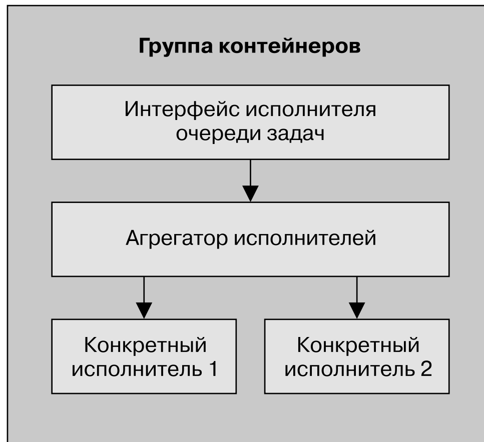


Рис. 11.5. Агрегирующий паттерн Multi-Worker, реализованный в виде группы контейнеров

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

Резюме

В этой главе мы начали обсуждение систем пакетной обработки с представления *потока задач* как последовательности дискретных действий, которые все вместе решают некоторую задачу, например обрабатывают заказы или анализируют видео с помощью моделей машинного обучения. В этой главе показано, как можно использовать контейнеры для создания дискретных рабочих процессов с контейнерами-прицепами, обеспечивающими общую оркестрацию, и контейнерами-исполнителями, осуществляющими конкретную обработку. Поток задач является ключевой частью большинства распределенных пакетных систем. Его общий паттерн легко адаптировать и расширять для удовлетворения конкретных потребностей рабочей нагрузки.

ГЛАВА 12

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. 12.1.

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. Кроме того, несмотря на то что первоначальная конструкция очереди

задач может быть простой, добавление обработки ошибок и других непредвиденных условий быстро усложняет реализацию очереди.

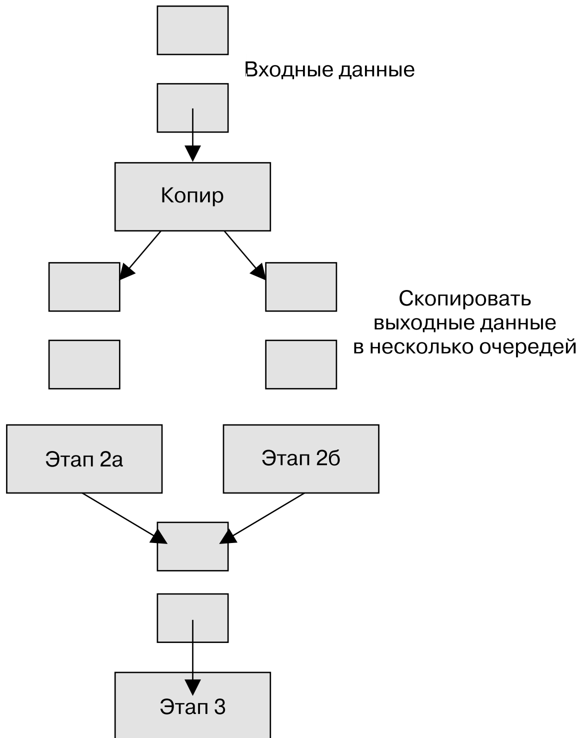


Рис. 12.1. В данном потоке задачи копируются в несколько параллельно обрабатываемых очередей (этапы 2а, 2б), а затем снова объединяются в общую очередь (этап 3)

Паттерны событийно-ориентированной обработки

Помимо простейшей очереди задач, описанной в предыдущей главе, существует ряд паттернов объединения нескольких очередей вместе. Простейший из них, заключающийся в передаче выходных данных одной очереди на вход другой, не требует рассмотрения.

Разберем 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 необходимо выполнить несколько различных 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 можно отвести отдельную очередь, но на вход им будут подаваться одни и те же данные. Применение паттерна Copier для перекодирования видео схематически изображено на рис. 12.2.

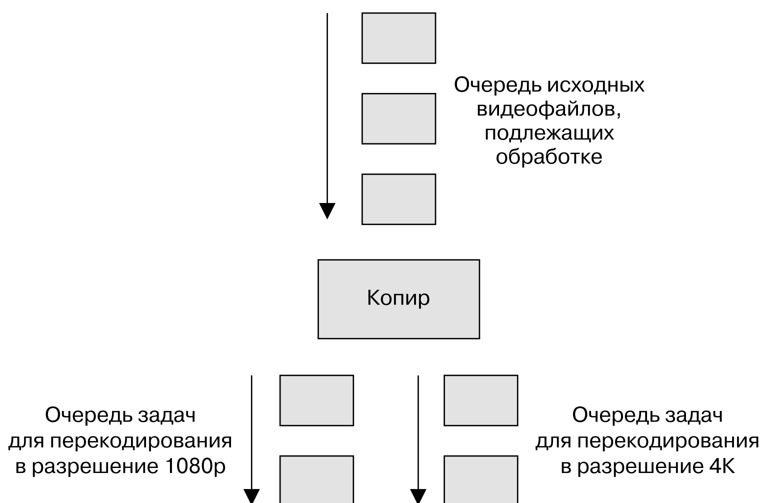


Рис. 12.2. Применение паттерна Copier для перекодирования видео

Паттерн Copier также можно использовать для обмена избыточной вычислительной мощности на повышенную надежность или производительность. Представьте, что у вас есть сервис, обрабатывающий задачу обычно за одну секунду, но иногда на обработку может уходить целая минута (замедление в 60 раз). Если каждую задачу передавать двум одинаковым сервисам и выбирать тот результат, который будет получен первым, то вероятность пострадать от такого замедления значительно уменьшится. Аналогично если сервис выходит из строя в 1 % случаев, то вы можете перейти от системы с двумя девятками (надежность 99 %) к системе с четырьмя девятками (надежность 99,99 %), просто передавая каждую задачу двум сервисам. Очевидно, что в обоих случаях лучшим решением в долгосрочной перспективе будет усовершенствование системы и повышение ее надежности с применением инженерных решений. Но иногда, когда нет возможности привлечь инженеров или требуется быстрое временное решение, полезно иметь возможность просто выделить дополнительные ресурсы для устранения проблемы.

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. 12.3.

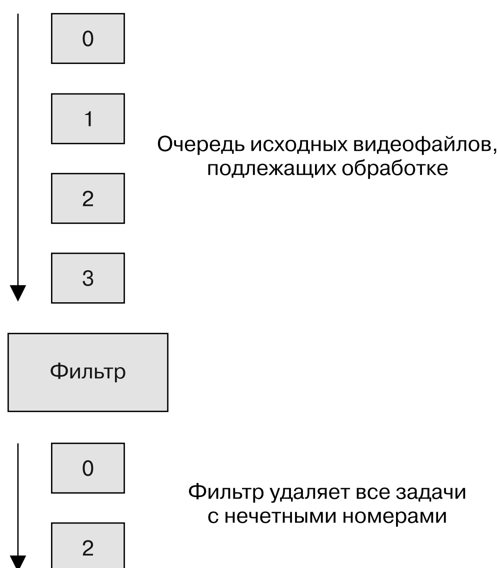


Рис. 12.3. Пример реализации паттерна Filter, когда удаляются все нечетные задания

Паттерн Splitter

Иногда может понадобиться не просто отфильтровать и отбросить ненужные задачи, а разделить их на два подмножества, каждое отправить в свою очередь. Для этого нужен контейнер-делитель. Цель делителя, как и фильтра, — вычислить значение некоторого критерия. Задачи, не удовлетворяющие этому 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 направлять каждый заказ в одну из двух очередей — для отправки уведомления по электронной почте или текстовым сообщением. Контейнер-делитель может также играть роль копира, отправляющего задачи в несколько очередей. Такое может

понадобиться, если при оформлении заказа пользователь выбрал оба типа уведомлений. Интересно также отметить, что делитель можно реализовать в виде комбинации копира и двух фильтров. Делитель, однако же, позволяет решить данную задачу более компактным образом. В этом отношении его можно сравнить с логическим вентилем XOR (исключающее ИЛИ), который можно реализовать с помощью вентилей OR/NOT/AND, но удобнее рассуждать о нем как об одной операции XOR. Пример использования паттерна Splitter для рассылки уведомлений о доставке приведен на рис. 12.4.

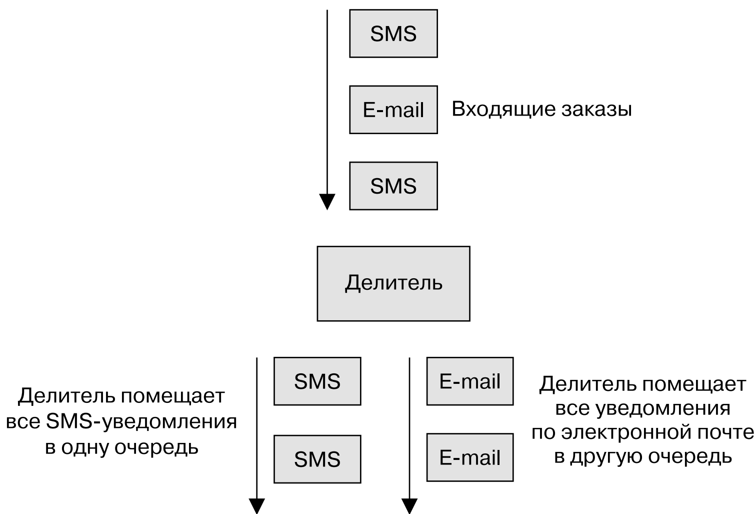


Рис. 12.4. Пример реализации паттерна пакетной обработки Splitter для рассылки уведомлений о доставке посредством двух очередей задач

Паттерн Sharder

Паттерн Sharder — несколько более общая форма паттерна Splitter. Цель шардера потока задач, как и шардирующего сервера, рассмотренного несколькими главами ранее, — разделить поток 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 потока задач отказ одной из очередей из-за

неудачного обновления, отказа инфраструктуры или по любой другой причине повлияет лишь на часть пользователей вашего сервиса.

Представьте, к примеру, что вы неудачно обновили контейнер-исполнитель, в результате чего все его экземпляры отказали и очередь перестала обрабатывать задачи. Если задачи обрабатываются только одной очередью, то сервис окажется недоступным для всех пользователей. А если поток задач шардируется на четыре части, то у вас будет возможность выполнить развертывание контейнера-исполнителя поэтапно. Если на первом этапе произойдет отказ, то при шардировании на четыре части он 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 потока 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. 12.5.

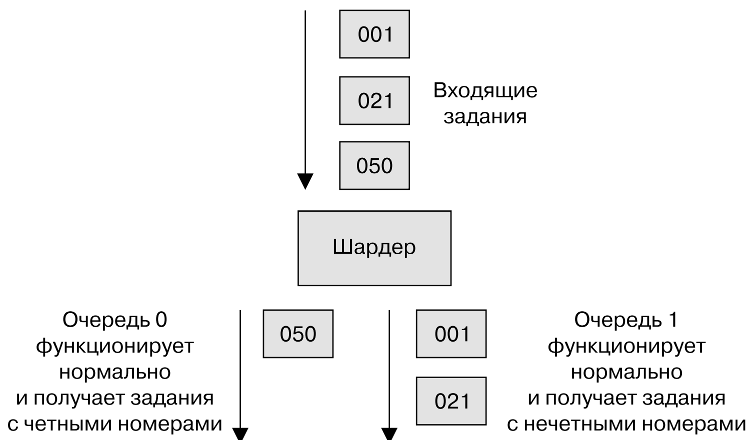


Рис. 12.5. Пример реализации шардированной очереди, работающей в штатном режиме

Если в силу отказов количество работоспособных шардов уменьшится, то алгоритм шардирования динамически перестроится и будет распределять задания только между работающими шардами, даже если останется только одна очередь. Это показано на рис. 12.6.

Рис. 12.6. При отказе одной из очередей задач оставшиеся переходят в другую очередь

Паттерн Merger

Последним среди паттернов событийно-ориентированных систем пакетной обработки рассмотрим паттерн *Merger*. Его действие противоположно действию паттерна Copier — он объединяет две очереди в одну общую. Допустим, в вашем проекте много разных репозиториев исходного кода, коммиты в которые происходят одновременно, и вам нужно выполнить тестирование и сборку для каждого из них. Создавать отдельную инфраструктуру сборки для каждого репозитория — плохо масштабируемое решение. Каждый из репозиториев можно смоделировать в виде очереди задач, служащей источником

задач-коммитов. Все эти источники задач можно объединить в общий 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. 12.7.

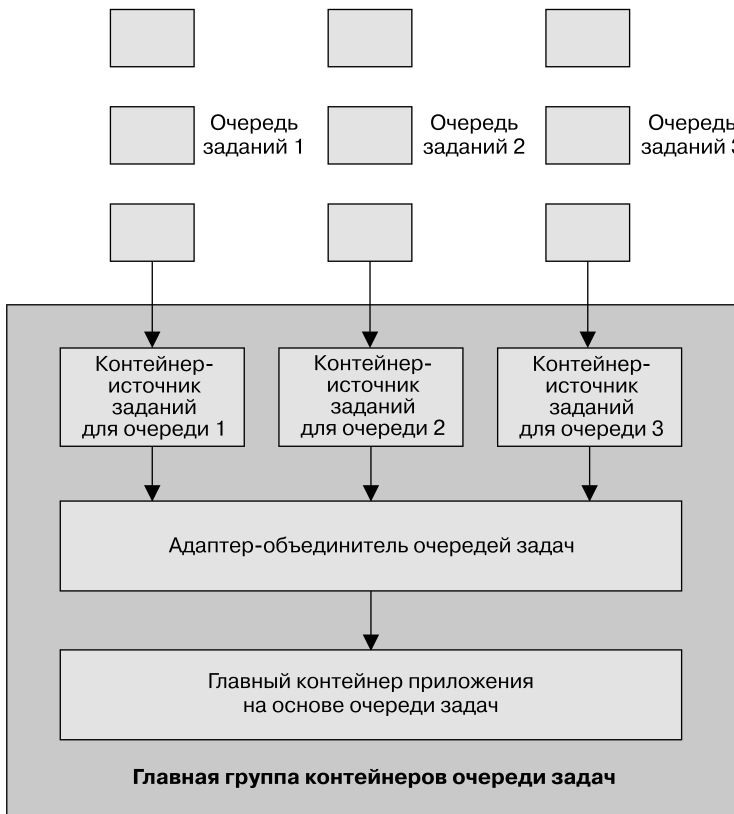


Рис. 12.7. Использование нескольких уровней контейнеров для объединения нескольких очередей задач в одну общую

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 SMS-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. 12.8.

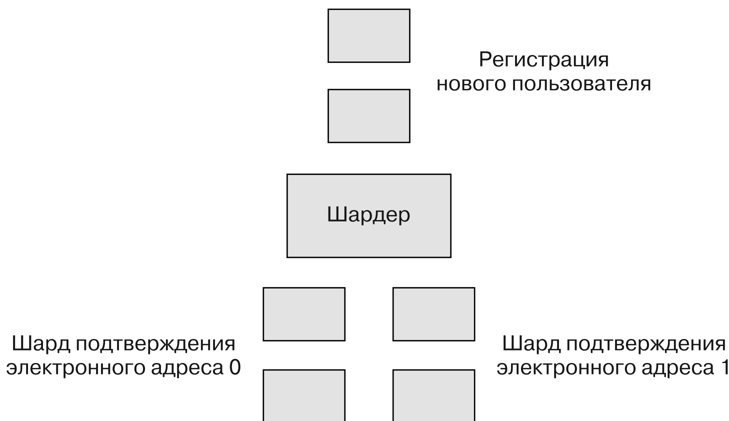


Рис. 12.8. Первый этап потока задач по регистрации нового пользователя

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 SMS-уведомления. Соответствующие очереди задач подписывают пользователей на уведомления по электронной почте и/или SMS.

Оставшаяся часть потока задач приводится на рис. 12.9.

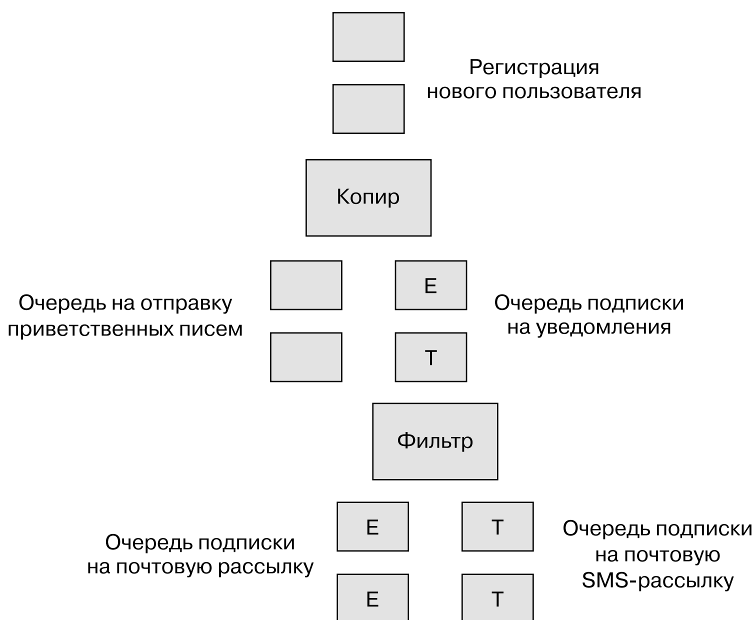


Рис. 12.9. Очередь задач по уведомлению пользователей и рассылке приветственных электронных писем

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 определенный каталог в локальной файловой системе и затем на каждом этапе обработки проверять появление новых задач в этом каталоге. 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 подразумевает использование контейнера под управлением оркестратора Kubernetes и менеджера пакетов Helm.

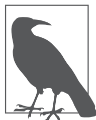
Helm — менеджер пакетов для Kubernetes, упрощающий развертывание и управление готовыми приложениями наподобие Kafka. Если у вас еще не установлен инструмент командной строки helm, загрузить его можно с сайта http://helm.sh.

Инструмент `helm` после установки необходимо инициализировать. В процессе инициализации `helm` разворачивает в кластере компонент под названием `tiller` и устанавливает ряд шаблонов в 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
```



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

После установки и запуска Kafka можно создавать темы и публиковать в них события. 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 0 1 2; 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.

Устойчивость и производительность в очередях задач

До сих пор мы старались оптимизировать конструкцию очередей заданий под конкретные нужды. Это важная часть проектирования. Чем ближе конструкция системы соответствует решаемой задаче, тем легче ее разрабатывать, обновлять и расширять. Однако не менее важным аспектом проектирования является прогнозирование и проектирование систем с учетом производительности и надежности.

Во всех предыдущих проектах мы предполагали, что все задания в системе примерно идентичны и каждый поток обрабатывается одинаково надежно. К сожалению, в реальном мире все эти предположения часто оказываются необоснованными. Большинство систем, кроме очень-очень необычных, обрабатывают смесь разных заданий. Разные видеофайлы, разные электронные письма, разные сборки — очереди постоянно обслуживают похожие, но разные задачи.

Перехват заданий

В результате различий на обработку некоторых заданий требуется больше времени. Иногда это обусловлено особенностями самих заданий (например, какие-то видеофайлы просто больше в размерах), но чаще причина кроется в проектных решениях, принятых при

разработке, которые лучше или хуже соответствуют конкретному заданию. Мы можем оптимизировать наш код для обработки большого количества файлов в небольшом количестве каталогов, а задание может, наоборот, заключаться в обработке небольшого количества файлов в глубоком дереве каталогов. Часто задержка отдельных заданий может не иметь значения — если обычно они попадают в разные очереди благодаря шардингу, то общее среднее время обработки обычно согласуется с нашими предположениями. Но, к сожалению, нередко средняя производительность оказывается хуже запланированной и все задания, требующие длительной обработки, оказываются в одной очереди. Когда это происходит, входные данные для очереди накапливаются и задержка увеличивается, как в медленной очереди в кассу в продуктовом магазине. Как и в продуктовом магазине, лучшим выходом из ситуации является переход из длинной очереди в другую, более короткую очередь. Этот алгоритм называется «перехват заданий» (work stealing).

Идея алгоритма перехвата заданий заключается в том, что любой исполнитель, пока в его очереди есть работа, действует как обычно. Но когда очередь опустошается, то вместо того, чтобы приостановиться до момента появления очередного задания, исполнитель перехватывает задание из конца самой длинной очереди любого другого исполнителя. При таком подходе максимальная длина любой очереди сводится к минимуму, потому что исполнители, быстро справляющиеся со своими заданиями, забирают задания у других исполнителей с более длинными очередями. Задания извлекаются из конца очереди, потому что это извлечение можно сделать безопасно, не заботясь о том, что одно и то же задание перехватят несколько исполнителей.

Ошибки, приоритеты и повторы

Нередко в системе с очередями возникают другие проблемы, когда происходят ошибки при обработке самих заданий. Первая проблема — сбой исполнителя в середине задания. В худшем случае задание, извлеченное из очереди, просто теряется. К счастью, большинство систем очередей поддерживают семантику сокрытия обрабатываемого задания, когда задание, принятое в обработку, скрывается от других

исполнителей, но на самом деле остается в очереди. По завершении обработки отправляется второе сообщение «завершено», которое навсегда удалит задание из очереди. Обычно очередь сообщений имеет тайм-аут, по истечении которого, если задание не было «завершено», оно возвращается в начало очереди, чтобы другие работники могли его забрать.

К сожалению, обработка таких ошибок иногда приводит к дополнительным осложнениям. Что, если первый исполнитель не потерпел сбой, а просто работает медленно? Тогда в системе будет два исполнителя, обрабатывающих одно и то же задание. Самое простое решение для противостояния таким ситуациям — сделать каждое задание идемпотентным. В таком случае обработка задания вторым исполнителем выльется лишь в пустую трату вычислительных ресурсов, но не вызовет дополнительных ошибок. Если нет возможности сделать задания идемпотентными, то можно прибегнуть к различным методам распределенной блокировки, обсуждавшимся в предыдущих главах. Важный вывод заключается в том, что при разработке исполнителей всегда следует предполагать, что за обработку одного и того же задания могут взяться сразу несколько исполнителей.

Если предположить идемпотентность заданий, то означает ли это, что мы защищены от сбоев исполнителей? К сожалению, нет. Представьте, что сбой исполнителя происходит не из-за случайных ошибок, а из-за каких-то особенностей самого задания. Представьте, что есть некое «ядовитое» задание, который приводит к сбою исполнителя в 100 % случаев. В такой ситуации, когда один исполнитель восстанавливается после сбоя, «ядовитое» задание подхватывается другим исполнителем, который снова терпит сбой, заставляя другого исполнителя подхватить это же задание, и т. д., пока все процессы не начнут терпеть сбой. Даже если вам повезет и ваши исполнители быстро восстановятся, «ядовитое» задание может привести к потере значительной части производительности и серьезно замедлить работу очереди.

Первое решение в исправлении проблемы «ядовитых» заданий — отслеживать количество попыток обработать их. Затем можно использовать экспоненциальное увеличение тайм-аута в обработке

«ядовитого» задания и по достижении некоторого порога удалить его из очереди навсегда. Как правило, экспоненциальная задержка решает проблему «ядовитых» заданий.

К сожалению, повторная постановка задания в очередь для обработки создает дополнительные проблемы. Чтобы понять причину, рассмотрим случай, когда исполнители аварийно завершаются по причинам, не связанным с заданием. Возможно, нисходящая зависимость возвращает неверные данные и вызывает сбой. В такой ситуации очередь фактически останавливается и в ней накапливается все больше и больше заданий. Поскольку ни один исполнитель не может выполнить задание, даже описанный выше перехват заданий не поможет. Предположим, что ситуация разрешится сама собой и исполнители смогут обработать задание, как в аналогии с пробкой на дороге после аварии — за это время в очереди может накопиться много заданий и вновь прибывшие запросы продолжат испытывать значительную задержку. Это часто особенно плохо в ситуациях, когда старые задания оказываются неактуальными и *желательно*, чтобы новые задания были выполнены вовремя.

Для решения проблемы невыполненных заданий часто полезно иметь две очереди. Первая очередь используется для всех заданий, которые еще ни разу не обрабатывались, а вторая — для «повторных попыток», то есть для любых заданий, обработка которых хотя бы один раз завершилась неудачей. Если использовать этот подход с двойной очередью, в очереди повторных попыток может скопиться немало заданий, ожидающих повторной обработки, но новые задания больше не будут стоять в конце очереди, позади заданий, потерпевших неудачу. Они будут иметь равные шансы выполниться вместе с заданиями из очереди повторных попыток. Этот подход является примером решения с общей приоритетной очередью, где разные очереди имеют разный приоритет и задания извлекаются из очереди с более высоким приоритетом, пока она не опустеет, а затем из очереди со следующим наивысшим приоритетом и т. д. Так можно гарантировать, что производительность будет направлена туда, где она больше всего нужна для достижения целевых показателей по величине задержки.

Резюме

В этой главе была представлена идея очередей заданий. Рабочий процесс разбивает задания на несколько очередей для достижения более сложной цели. Мы также мы рассмотрели несколько паттернов построения рабочих процессов, от копирования до фильтрации, шардирования и объединения. Наконец, обсудили такие важные аспекты, как производительность и надежность в условиях медленной работы и сбоев исполнителей. Используя эти паттерны и подходы к обработке ошибок, любую сложную работу можно реализовать в виде простого и понятного рабочего процесса, при котором надежно и эффективно обрабатываются данные.

ГЛАВА 13

Координированная пакетная обработка

В предыдущей главе мы рассмотрели паттерны разделения и объединения 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. 13.1.

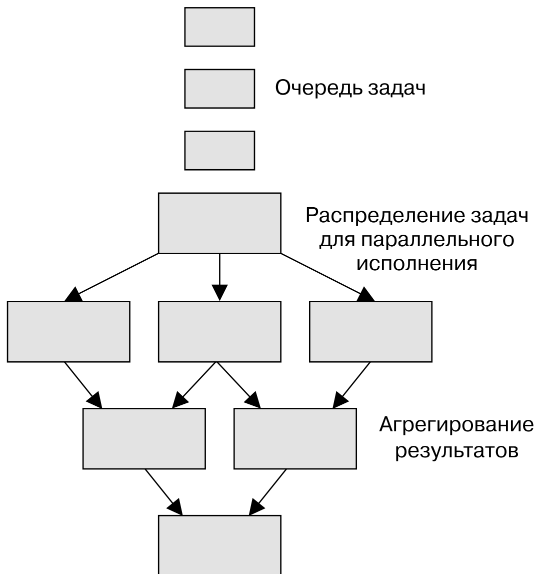


Рис. 13.1. Обобщенная пакетная система распределения задач и агрегирования результатов

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.

Паттерн Join (барьерная синхронизация)

В предыдущих главах мы рассмотрели паттерны распределения заданий 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 попросту 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.

В качестве конкретного примера необходимости применения паттерна *Join* рассмотрим создание цифрового мультипликационного фильма. Кадры фильма являются независимыми заданиями. Их можно обрабатывать по отдельности, без привязки к другим кадрам в фильме. Но давайте рассмотрим преобразование кадров фильма в видео длительностью 1 секунда. Нам определенно нужно получить все тридцать кадров, составляющих эту 1 секунду, прежде чем преобразовать их в видео.

Нам нужен другой примитив пакетной обработки данных, более строгий и координирующий. Таким примитивом является 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. 13.2.

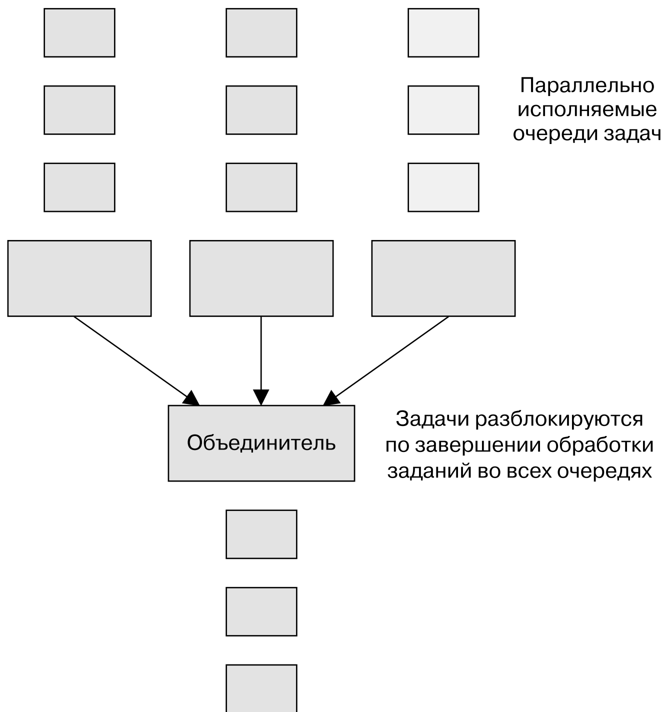


Рис. 13.2. Паттерн пакетной обработки Join

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, как обсуждалось в главе 12. Увеличение задержки можно компенсировать, пожертвовав вычислительными ресурсами ради скорости и дублируя обработку всех заданий, чтобы уменьшить влияние задержки.

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 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, результатом свертки будет единственный элемент выходных данных с подсчитанными количествами различных слов, присутствующих в книге. Конечно, количество запусков свертки влияет на общую производительность пакетной обработки. Слишком большое количество операций свертки может вызвать значительную потерю производительности, а слишком малое начинает напоминать синхронизацию в паттерне Join.

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 определите 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, 4 000 000) и (Нортгемптон, 25 000), и просуммировать их. В результате получится новый выходной элемент (Сиэтл-Нортгемптон, 4 025 000). 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%
```

Как показали предыдущие примеры, все эти гистограммы можно объединить 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 маркировка и обработка изображений

Чтобы понять, как использовать координированную пакетную обработку для выполнения сложных пакетных задач, рассмотрим задачу маркировки 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 — поиск 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 11 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. 13.3.

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 предыдущем разделе. С его помощью мы объединим шардированные очереди задач в общую очередь, которая освободит элементы для дальнейшей обработки только после того, как все шарды завершат работу.

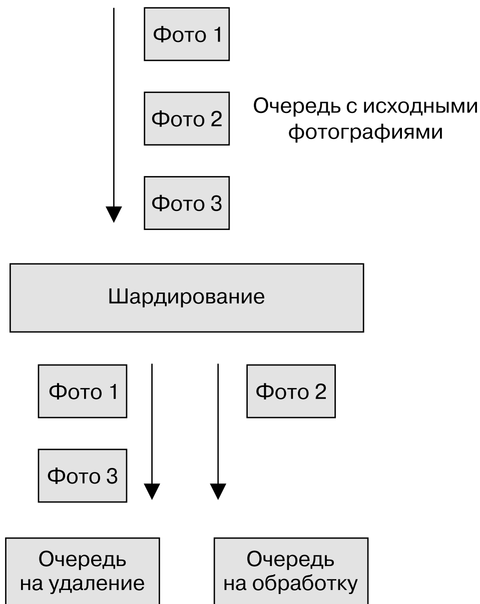


Рис. 13.3. Шардированная очередь с несколькими шардами для размытия изображения

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 для удаления оригиналов;
- очередь задач для определения цвета и марки машины.

На рис. 13.4 схематически изображены упомянутые стадии обработки.

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 для распознавания местоположения и типа транспортного средства, а второй — для определения его цвета. Чтобы организовать слияние, снова воспользуемся паттерном Multi-Worker, рассмотренным в главе 11. 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.

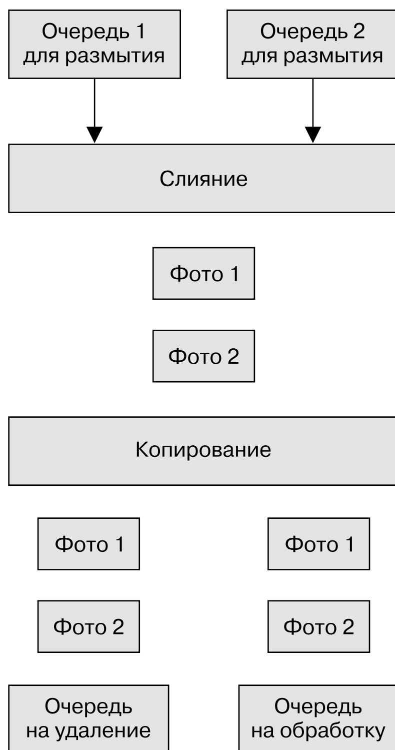


Рис. 13.4. Этапы слияния и копирования очередей, удаления исходных изображений и распознавания образов

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 в примере с подсчетом. Стадия свертки в качестве результата выдает итоговое количество образов и цветов, найденных во всем множестве снимков.

Резюме

В этой главе описываются приемы *координированной пакетной обработки*, применяемые для синхронизации рабочих процессов на основе предварительных условий или для объединения выходных данных из предшествующих рабочих процессов в общий результат. Паттерн Join используется в качестве барьерного синхронизатора. Он гарантирует готовность всех необходимых выходных данных перед переходом к следующему этапу. Паттерн Reduce объединяет выходные данные из нескольких шардов, генерируя конечный результат. Суммирование и гистограмма — это особые разновидности операции свертки, которые удобно использовать для получения определенных статистических метрик из имеющихся данных. Все вместе эти паттерны для координации действий в пакетной обработке могут гарантировать построение рабочего процесса, необходимого для любых пакетных операций или вычислений, которые могут вам понадобиться.

Часть V

Универсальные концепции

До сих пор основное внимание мы уделяли паттернам создания приложений, от отдельных узлов до сложных распределенных систем. Но в наших системах есть еще кое-что общее для всех приложений. Некоторые из таких универсальных концепций мы рассмотрим в следующих главах. В главе 14 обсуждаются вопросы улучшения наблюдаемости приложений посредством журналирования, мониторинга и рассылки уведомлений. Наблюдаемость имеет большое значение, так как позволяет понять, правильно ли работает приложение. Глава 15 посвящена ИИ, который коренным образом меняет наш взгляд на приложения и пользовательские интерфейсы. Наконец, в главе 16 мы исследуем распространенные ошибки и погрешности, которые с печальной регулярностью происходят в распределенных системах. Знания, которые вы получите в этих главах, помогут вам создавать более надежные и интеллектуальные приложения, независимо от их назначения.

ГЛАВА 14

Паттерны мониторинга и наблюдаемости

Одно из основных отличий распределенных систем от клиентских приложений заключается в том, что распределенные системы обычно реализуют сервисы. Сервисы всегда работают и всегда доступны пользователям по всему миру во всех часовых поясах. А так как эти системы работают в круглосуточном режиме, мониторинг и наблюдаемость становятся критически важными для обеспечения высокой надежности. Чтобы достичь высокой надежности, мы должны замечать проблемы раньше, чем их заметят клиенты, а чтобы быстро исправить их, мы должны понимать, как работает система. В данной главе основное внимание уделяется передовым практикам мониторинга и обеспечения наблюдаемости, помогающим в этом.

Основы мониторинга и наблюдаемости

Прежде чем углубиться в детали реализации мониторинга и наблюдаемости, полезно ознакомиться с ключевыми концепциями, на которых основывается любое решение мониторинга и наблюдаемости.

Существует четыре ключевые концепции, образующие фундамент для наших решений:

- журналирование;
- метрики;
- оповещение;
- трассировка.

Далее мы подробно рассмотрим каждую из них.

Каждый, кто участвовал в создании даже самой маленькой системы, почти наверняка реализовал журналирование, даже не осознавая

этого. Самая простая версия журналирования — скромная инструкция `printf`. Конечно, существует множество более сложных способов журналирования, но все они служат той же цели, что и эта инструкция вывода. А именно, они подсказывают, что определенный фрагмент программного кода выполнился, и фиксируют данные, так или иначе связанные с выполнением этого кода. В журнал записываются данные, помогающие понять, как выполняются те или иные этапы.

Другие ключевые концепции, дополняющие журналирование, — это фиксация метрик или мониторинг. Если журналирование связано с выполнением конкретных фрагментов кода, то фиксация метрик обычно заключается в агрегировании данных, характеризующих некоторую последовательность запросов, например количество вызовов некоторой функции за последнюю минуту или среднюю продолжительность обработки вызовов. Журналирование дает нам представление о выполнении конкретных участков кода, а метрики — видение работы сервиса в целом.

Цель журналирования и фиксации метрик — помочь нам понять, где кроются проблемы, когда они возникают. Если бы все свое время мы тратили на просмотр графиков метрик и журналов, то мы всегда имели бы полное представление о происходящем, но в реальной жизни бóльшую часть времени мы тратим на выполнение других работ. Следовательно, нам нужны *оповещения*, сообщающие, что возникли какие-то проблемы и мы должны оторваться от повседневной работы, чтобы заглянуть в журналы или исследовать метрики. По сути, оповещения — это правила, которые применяются к журналам или к метрикам и запускают события. Такие события обычно приводят к отправке уведомлений, призванных привлечь наше внимание к проблеме (или потенциальной проблеме) в системе. Создание высококачественных оповещений — ключевой аспект создания надежных систем.

Последний универсальный компонент касается распределенного характера наших систем. Одна из самых больших сложностей при построении распределенной системы — это понимание особенностей совместной работы всех ее частей или, что еще сложнее, понимание причин, почему они могут выйти из строя при определенных обстоятельствах. Наивно полагать, что каждый отдельный запрос к микросервису уникален и индивидуален. В действительности же каждый запрос, отправленный некоторому микросервису, является

частью цепочки запросов, порождаемой одним большим запросом. *Трассировка* — это процесс перестройки контекста, помогающий на основе отдельных и уникальных запросов в каждом микросервисе получить более широкое представление об обработке большого пользовательского запроса. Это более широкое представление, в свою очередь, позволяет понять (например), почему запрос конкретного пользователя выполняется медленно или вообще терпит неудачу. Трассировка распространяется на все части распределенной системы и реконструирует видение пользователя.

В следующих подразделах подробнее исследуем каждый из этих компонентов и посмотрим, как их использовать для создания надежной системы.

Журналирование

Итак, мы выяснили, как журналировать информацию, и перед нами тут же встает следующий вопрос: какой объем информации журналировать? Объем журналов, или «журнальный спам», часто оказывается значительной проблемой с точки зрения их исследования или даже просто затрат на хранение и поиск по всем этим журналам.

Важно понимать, что ценность журналирования сильно зависит от обстоятельств. В период отладки конкретной проблемы в конкретном компоненте подробное журналирование его работы может иметь решающее значение. С другой стороны, в период промышленной эксплуатации, когда все компоненты отлажены и работают безупречно, значительная часть информации в журнале оказывается бесполезным шумом, за исключением тех журналов, которые порождают оповещения из-за системных ошибок.

По этим причинам только в самых простых распределенных системах для журналирования используются простые инструкции вывода. Все остальные используют некоторую библиотеку журналирования. Такие библиотеки помогают значительно упростить работу с большим количеством журналов. Основная возможность, предоставляемая такими библиотеками, — добавление отметки времени в записи. Исследуя проблему, мы часто пытаемся выяснить, что произошло и в какой последовательности, а без единой хронологии будет трудно узнать (например), сколько времени прошло между разными событиями, зафиксированными в журнале. Помимо отметки времени, большинство

библиотек журналирования позволяют добавлять другую контекстную информацию. В любой многопоточной системе запросы могут обрабатываться параллельно, и поэтому в журнал может заноситься множество идентичных записей. Чтобы можно было различать эти записи и понимать, как они связаны с конкретным запросом, в них может добавляться дополнительный контекст — идентификаторы потоков выполнения, идентификаторы пользователей или идентификаторы запросов. Эти дополнительные сведения помогают различать в остальном идентичные записи и выполнять фильтрацию, чтобы оставить только сведения, касающиеся конкретного запроса. Последняя и часто самая ценная возможность, предоставляемая библиотеками журналирования, — это поддержка уровней журналирования. Уровни журналирования позволяют задать, какие записи должны попадать в журналы. Разные библиотеки могут поддерживать разное количество уровней, но обычно в их число входят следующие.

Debug

Подробное журналирование, как правило, полезное только на этапе отладки.

Info

Журналирование в целом полезных сведений о работе сервиса, не связанных с исключительными ситуациями.

Warning

Произошло что-то тревожное, но не обязательно фатальное для всего приложения.

Error

Произошло что-то очень плохое, и кто-то должен как можно скорее разобраться с этим.

Fatal

Произошло что-то непоправимое, и приложение завершит работу после записи этого сообщения в журнал.

Поддержка уровней журналирования позволяет регистрировать в журнале только определенные типы сообщений, а также оставлять инструкции подробного журналирования в коде на случай, если они понадобятся, и при этом не переполнять журналы множеством ненужных сообщений. Обычно в приложениях устанавливается уровень Info

или Warning, в зависимости от принятого стиля журналирования. Сообщения с уровнем Debug активируются по условию и зарезервированы для отладки конкретных проблем.

Помимо контекста запроса, ценным контекстом для библиотеки журналирования является место, откуда произведена запись в журнал. Обычно системы журналирования понимают, какой класс или компонент производит запись. Это обеспечивает возможность *условного журналирования*, когда динамически активируется уровень Debug для определенных классов или других компонентов на работающих серверах. Условное журналирование позволяет реагировать на сообщение об инциденте или проблему, замеченную определенным пользователем, без переключения на подробное журналирование для каждого запроса. При использовании условного журналирования следует проявлять некоторую осторожность, так как это может значительно увеличить задержку в обработке запросов конечного пользователя. Нет ничего хуже, чем вызвать еще серьезный инцидент или даже сбой в процессе расследования сбоя.

Чтобы сделать журналирование полезным, необходимо иметь четкое понимание главной его цели. В конечном счете журналы дают представление о работе сервиса или приложения. Часто программисты шутят, что для того, чтобы понять, что журналировать, нужно вспомнить, что было упущено, когда проблема отлаживалась в первый раз. То есть настройка журналирования часто является ретроспективным упражнением: «Эх! Надо было зафиксировать в журнале это и вот это...» Разумеется, что самые полезные события в журналах — это исключительные или ошибочные ситуации. Каждая ошибка или исключение, вероятно, должны фиксироваться, если у вас нет полной уверенности в обработке ошибок в коде (а иногда даже и в этом случае). Помимо ошибок, также желательно фиксировать в журнале возникающие странности. Например, некоторые запросы могут выполняться очень медленно. Метрики (см. подраздел «Метрики» далее) могут говорить вам, что 99 % запросов обрабатывается чрезмерно долго, но без журналов, сообщающих, какие этапы обработки выполняются исключительно медленно, будет трудно решить, какие действия следует предпринять. С другой стороны, простое журналирование последовательности выполняемых операций («вызвана функция `foo`») редко бывает полезным, и такие сообщения просто заполонят ваши журналы. Реализуя журналирование, попытайтесь

представить самую сложную проблему, которую может потребоваться отладить в этой конкретной части кода, и запишите в журнал все, что может понадобиться для ее разрешения.

Метрики

В отличие от журналов с их фокусом на выполнении отдельных участков кода метрики мониторинга предназначены для сбора статистики, характеризующей работу сервисов с течением времени. В этом подразделе основное внимание мы сосредоточим на системе мониторинга Prometheus (https://prometheus.io), которая стала фактическим отраслевым стандартом. Другие решения для мониторинга предлагают схожие возможности.

Данные мониторинга обычно фиксируются постоянно, и с этой точки зрения у вас есть возможность видеть как мгновенное состояние вашего приложения, так и тенденции в его изменении с течением времени.

При мониторинге приложения обычно фиксируется три типа данных:

- гистограммы;
- счетчики;
- значения.

Гистограммы описывают распределение значений в некотором временном интервале. Они позволяют увидеть как средние, так и экстремальные показатели работы. Например, гистограмма задержки может показать величину задержки среди 10 % самых быстрых запросов, среднюю задержку и величину задержки среди 10 % самых медленных запросов. Такая информация поможет быстро оценить, как работает приложение.

Счетчики — это монотонно увеличивающиеся значения, позволяющие отслеживать некоторые величины. Типичным примером счетчика может служить общее количество запросов, обработанных сервисом. Очевидно, что это число может только увеличиваться со временем, потому что запросы продолжают и продолжают поступать. Счетчики также часто преобразуются в метрики, которые являются производными первого порядка от счетчиков и представляют собой скорость изменения соответствующего счетчика. Например, метрика,

основанная на счетчике запросов, может сообщить количество запросов, обрабатываемых в секунду. Счетчики по своей природе являются целыми неотрицательными числами.

Последний тип метрик — значения. Это числа, которые могут принимать любое значение, а также увеличиваться и уменьшаться с течением времени. Типичным примером значений является потребление ЦП и памяти. Очевидно, что потребление ЦП и памяти может меняться с течением времени и увеличиваться или уменьшаться в зависимости от нагрузки на сервис. Значения наиболее полезны при наблюдении за их изменением с течением времени, поскольку они могут характеризовать общее поведение системы. Постоянное увеличение потребления памяти может указывать на ее утечку. Периодически скачущая нагрузка на ЦП может говорить о том, что какой-то фоновый процесс потребляет неоправданно много вычислительных ресурсов.

На данный момент вам должно быть ясно, что главная ценность метрик мониторинга — это общая картина, которую они дают с течением времени. Действительно, большинство данных мониторинга называют *временны́ми рядами*, представляющими некоторые значения с течением времени. Для эффективного хранения и извлечения этих значений были разработаны специальные базы данных временных рядов. Система Prometheus тоже поставляется с простой базой данных временных рядов, но обычно ее интегрируют со специализированной базой данных. Часто хорошим вариантом является интеграция системы мониторинга Prometheus с облачными средствами мониторинга, предлагаемыми в виде сервисов. В настоящее время большинство облачных средств мониторинга, например Azure Monitor, поддерживают прием метрик из Prometheus.

С точки зрения получения метрик Prometheus является системой, активно собирающей метрики. Prometheus периодически посылает запросы вашему приложению на получение текущих значений всех метрик и сохраняет полученные сведения в базе данных временны́х рядов для последующего использования. Мониторинг на основе активного извлечения метрик хорошо подходит для случаев, когда приложения работают постоянно, но как собрать метрики из приложений пакетной обработки и других задач, выполняющихся периодически, которые могут запуститься, быстро обработать задание и завершиться до того, как Prometheus начнет очередной цикл опроса?

Для таких заданий нужно реализовать *принудительную отправку метрик*. К счастью, у Prometheus есть родственный проект, Prometheus Pushgateway, реализующий такой подход. При запуске Pushgateway запускает выделенный сервер, который Prometheus сможет опросить. Ваши приложения могут отправлять метрики в этот шлюз, чтобы они фиксировались, даже после остановки приложения. Сочетание мониторинга на основе активного опроса и принудительной отправки метрик (pull и push соответственно) позволяет получить полную картину работы приложения.

Базовый мониторинг запросов

Один из первых широко используемых типов мониторинга — наблюдение за количеством запросов, обслуживаемых приложением. Эта базовая статистика показывает, сколько людей (или других сервисов) использовало ваш сервис. Количество запросов — это пример метрики-счетчика. Счетчик запросов не может уменьшаться, потому что не существует такого понятия, как отрицательный запрос.

Однако на самом деле общее количество запросов не особенно интересно, гораздо интереснее скорость изменения общего количества запросов. Скорость изменения — это вторичная метрика, определяемая как разность между количеством запросов в момент времени t и количеством запросов в момент времени $t + 1$. Например, скорость запросов в минуту — это разность между общим количеством запросов прямо сейчас и общим количеством запросов минуту назад. Чтобы превратить эту метрику в график, нужно вычислить ее для каждой прошедшей минуты с интервалом 1 минута. Конечно, в этом примере единицу времени 1 минута я выбрал произвольно, на практике вы можете выполнить эквивалентные вычисления для любого нужного вам интервала времени, от миллисекунды до суток (и даже больше).

Пример с расчетом скорости запросов за миллисекунду демонстрирует одну из проблем, связанных с метриками скорости. Если сервер обслуживает менее 1000 запросов в секунду, то скорость запросов, вычисленная с точностью до миллисекунды, будет показывать резкие колебания между нулем и единицей (и, возможно, больше, если будут наблюдаться кратковременные всплески активности пользователей). Скорости изменчивы и сильно зависят от внешних

факторов. Поэтому часто полезнее использовать среднее значение скорости запросов за интервал времени. Усреднение скорости по времени сглаживает пики и спады и дает более реалистичную картину использования сервиса.

Общее количество запросов — хорошая метрика, но, чтобы по-настоящему понять характер работы системы, полезно иметь перед глазами более подробную информацию, связанную с каждым запросом. Одной из наиболее распространенных составляющих запроса является код ответа HTTP, возвращаемый пользователям. Для тех, кто пока не знаком с кодами ответов HTTP, отмечу, что они являются частью протокола HTTP и указывают, как сервер обработал запрос. Наиболее распространенные коды: 200 (OK), 404 (Not found — «не найдено») и 500 (Internal server error — «внутренняя ошибка сервера»), но есть и другие коды, описывающие разные ситуации. Метрики-значения в Prometheus могут иметь метки, добавляющие дополнительные детали к метрике. Учитывая эту особенность, можно организовать подсчет запросов с метками, содержащими код ответа. Благодаря этому можно узнать, например, общее количество успешно обработанных запросов (`code == 200`) или общее количество запросов, которые привели к ошибкам (`code == 500`); также можно вычислить частоту запросов каждого типа, применяя методы, описанные выше. Распределение общего количества запросов по коду ответа даст гораздо более полное представление о работе приложения. Можно также сгруппировать коды ошибок и посмотреть, сколько запросов привели к ошибке (коды, начинающиеся с 5) и сколько запросов были успешными (коды, начинающиеся с 2).

Помимо кода ответа еще одной важной характеристикой запроса является задержка — количество времени, потребовавшегося для его обработки. Может возникнуть соблазн представить задержку в виде метки при счетчике запросов, но этот прием не сработает. Метками может быть ограниченное число дискретных значений, тогда как величина задержки может принимать множество разных значений. Задержка запроса — это отдельная метрика, и она обычно моделируется в виде гистограммы. Используя метрику-гистограмму с информацией о задержках обработки запросов, можно понять, как приложение воспринимается средним пользователем, а также пользователями, испытывающими самые большие задержки. Так же как в случае с количеством запросов, задержку тоже можно снабдить

меткой с кодом ответа. Таким образом, вы можете не только увидеть средний опыт любого запроса, но и рассчитать среднюю задержку ошибки, запроса «не найден» или любого другого кода ответа HTTP.

Расширенный мониторинг запросов

Общее количество запросов, скорость и задержка являются наиболее распространенными и наиболее полезными метриками, однако существует ряд других, более специфических метрик, которые могут помочь глубже понять характер работы сервиса. Хорошим примером могут служить размеры запросов и ответов в байтах. Размер запроса можно использовать для выявления зависимости скорости обработки запроса от его размера (запросы большего размера могут дольше обрабатываться), а также для просмотра, как меняется размер с течением времени, чтобы выявить такие закономерности, как увеличение размеров запросов или ответов с течением времени. Как и задержка, размер запроса и ответа является примером метрик-гистограмм.

Иногда хорошими метрики делают детали, описывающие, как они вычисляются в приложении. Например, иногда полезно знать, сколько времени запрос провел в очереди и сколько времени было затрачено на его обработку. Такая информация особенно полезна при отладке проблем с медленной работой сервиса. Информация о больших задержках подскажет вам, что в приложении есть проблема и пользователи, вероятно, недовольны, а наличие метрик с временем ожидания в очереди и временем обработки поможет понять, на что направить свои усилия — на масштабирование (если время ожидания в очереди слишком велико) или на оптимизацию кода (время обработки слишком велико). Чтобы получить эти метрики, вам придется создать их в своем коде. В данном случае нужно добавить две новые метрики-гистограммы, одна из которых будет измерять время от момента получения запроса до начала обработки, а другая — от момента начала обработки до отправки ответа. Теперь наша гипотетическая система имеет три метрики-гистограммы, измеряющие задержки: общую задержку, задержку очереди и задержку обработки. Добавлять метрики совсем не сложно, а дополнительная информация, которую они несут, часто оказывается бесценной при отладке сервиса. Конечно, как и в любом деле, есть риск переусердствовать, и если вы пытаетесь зафиксировать задержку выполнения

каждой строки кода (или даже каждой функции), то, скорее всего, вы пытаетесь зайти слишком далеко.

Другим примером расширенных метрик, характеризующих обработку запросов, является разбивка по географическим местоположениям или клиентам. Эта информация может способствовать созданию более качественных продуктов или, например, выявлять оскорбительные запросы, поступающие от определенных пользователей или из определенных мест. Если вам достаточно грубой географической разбивки (например, по странам), то географию можно рассматривать как еще одну метку для метрики количества запросов, но чаще всего такие данные предпочтительнее извлекать из журналов, а не получать напрямую из метрик. Например, в любой большой системе клиентов слишком много, чтобы использовать их идентификаторы в качестве метки. Вместо этого лучше регистрировать идентификатор клиента с каждым запросом в журнале. Этот пример демонстрирует связь между журналированием и мониторингом. Мониторинг и метрики могут предоставлять агрегированные данные, а для получения более подробной информации лучше использовать журналирование и поиск в журналах, которые обеспечивают более медленный, но более содержательный доступ к данным. Бизнес-анализ, как правило, предъявляет менее строгие требования к задержке при получении информации, поэтому поиск в журналах может вполне соответствовать бизнес-требованиям.

Оповещение

До сих пор мы обсуждали различные способы извлечения информации из журналов и метрик приложения. Эти данные могут оказать существенную помощь в выявлении причин неправильного поведения системы, приведшего к сбою или другому инциденту. Но как узнать, что что-то произошло?

Очевидно, что первый источник информации о наличии проблемы в системе — это *жалоба клиента или пользователя*. Однако ожидание таких сообщений от клиентов не лучший выход как для пользователей, так и для того, кто принимает такие жалобы. Ничто не подрывает доверие клиента так сильно, как известие о том, что он был первым, кто заметил проблему. Работая с нашими системами, пользователи ожидают, что мы заметим (и исправим) проблемы до того, как они с ними столкнутся.

Поэтому лучший способ узнать о проблеме — это получить *оповещение*. Оповещение — это практика создания условий, при которых команда, ответственная за обслуживание, уведомляется отправкой сообщений о проблемах на пейджер или по электронной почте.

Базовое оповещение

Самая базовая форма оповещения основана на запросах метрик и оценке статических пороговых значений. Например, можно организовать отправку оповещений, если задержка в 90-м процентиле превысит полсекунды или если количество ответов с кодом HTTP 500 превысит 1 % от общего числа запросов. Как видите, всегда, когда происходит что-то необычное, было бы хорошей идеей сообщить об этом инженерам. Но как определить правильные подходы к оповещению в своей системе?

В простейшем случае оповещения следует рассматривать как определяющие цель уровня обслуживания (service level objective, SLO) для вашего приложения. Не имея оповещений, вы не заметите проблемы, поэтому необходимо задавать пороговые значения для оповещений, отделяющие номинальный и экстремальный режимы работы. Каждое приложение индивидуально, но никогда не лишне записать цели, гарантирующие «нормальный» пользовательский опыт при работе с вашим сервисом и правильные оповещения, срабатывающие, если эти цели не достигаются. Еще один фактор, который следует учитывать, — это опыт ваших дежурных инженеров. Постоянные оповещения приводят к выгоранию специалистов и появлению у них желания уволиться. Аналогично отправка ложных оповещений приведет к тому, что ваши дежурные инженеры перестанут реагировать на них (вспомните притчу о пастушке́, который шутки ради кричал: «Волки, волки!»). Поэтому старайтесь сбалансировать цели вашего сервиса с потребностями дежурных инженеров и определите правильные оповещения. Разработка оповещений — это непрерывная деятельность, не прекращающаяся на протяжении всего срока службы вашего сервиса. По мере добавления новых возможностей и, следовательно, расширения мониторинга вы также должны добавлять больше оповещений. По мере приобретения опыта работы с системой вы сможете настраивать оповещения, делая их более или менее жесткими в зависимости от потребностей пользователей или дежурных инженеров.

Оповещение об аномалиях

Базовое оповещение отлично подходит для большинства случаев, но иногда его недостаточно. Отличным примером может служить ошибка, которая приводит к полному отключению небольшой (1 %) подгруппы пользователей. С точки зрения статического оповещения ваша система функционирует в допустимых пределах, но для пользователей, которым не повезло попасть в эту подгруппу, ваш сервис полностью стал нефункциональным. Статическое оповещение предполагает, что все метрики одинаковы для всех пользователей и вызовов. Однако это не всегда так. Обнаружение аномалий — это прерогатива ИИ, который способен выявить ситуации, когда клиент А полностью лишился возможности связаться с сервисом, тогда как клиент Б работает без всяких проблем. Оповещение об аномалиях — сложная тема, но зачастую это единственный способ найти баланс между оповещением при возникновении проблемы и порождением множества ложных тревог. С ростом и усложнением вашей системы оповещение об обнаружении аномалий станет критически важным требованием.

Мониторинг и журналирование обеспечивают видимость вашего приложения. Оповещения подсказывают, что на проблему следует обратить внимание. Важно отметить, что не каждая метрика требует оповещения. Некоторые полезны только для отладки и визуализации. Однако качество работы вашего приложения будет определяться качеством ваших оповещений. Разрабатывайте оповещения с той же энергией, с какой вы разрабатываете новые возможности для клиентов.

Трассировка

Как было показано в предыдущих разделах, большинство современных приложений обрабатывают запросы, которые охватывают несколько машин и процессов. Методы, описанные ранее, помогут контролировать все эти компоненты по отдельности. К сожалению, такое «посервисное» представление показывает все запросы, обрабатываемые системой одновременно, но не показывает путь одного запроса через все микросервисы в системе. Трассировка запросов сопоставляет метрики из всех сервисов и дает эту сквозную перспективу.

Самый простой способ организовать трассировку — создать уникальный идентификатор для каждого запроса, поступающего в систему. Фактически роль идентификатора играет число, которое можно использовать для группировки записей из журналов, метрик и другой информации в приложении. Идентификатор может быть любым, если он уникально идентифицирует каждый запрос. Простейшим идентификатором может быть большое случайное число, а лучшим вариантом — хеш, вычисляемый по уникальным характеристикам запроса, таким как его путь RESTful и исходящий IP-адрес.

Снабдив каждый запрос уникальным идентификатором, вы должны добавлять его в журнальные записи или в метрики. Например, в системе журналирования, помимо отметки времени и уровня журналирования, вы также должны зарегистрировать идентификатор запроса. С этой целью идентификатор должен передаваться во все компоненты вашей системы вместе с вызовами API.

Если предположить, что запросы API выполняются с использованием протокола HTTP, то идентификатор можно встроить в заголовок запроса, специфичный для приложения, например `x-myapps-correlation-id`, и добавить код, который будет читать значение из этого заголовка и сохранять его в объекте контекста запроса. После того как будет организовано сохранение идентификаторов в журналах и метриках, вы сможете сгруппировать эту информацию и восстановить путь каждого запроса через всю систему.

Вы не ошибетесь, если подумаете, что реализация и правильная настройка регистрации идентификаторов — это очень сложная работа. К счастью, существует проект OpenTelemetry (https://opentelemetry.io), который может взять на себя большую часть этой работы. OpenTelemetry имеет SDK для всех популярных языков, что упрощает интеграцию трассировки запросов в приложения. Он следует платформенно независимой философии, что позволяет интегрировать его с многочисленными системами мониторинга и журналирования. Также с OpenTelemetry легко интегрируются многочисленные инструменты визуализации и отладки, что позволяет обеспечить богатые возможности для исследования. Почти во всех случаях интеграция с подобным проектом с открытым исходным кодом, поддерживаемым сообществом, для трассировки запросов является лучшей идеей, чем создание своего собственного решения.

Агрегирование информации

Внедрив мониторинг, вы быстро обнаружите, что он способен генерировать огромное количество информации. На этом этапе особую важность приобретает эффективное управление необходимой информацией, а также возможность быстро просеивать ее в поисках нужных сведений, касающихся текущей проблемы. Агрегирование позволяет группировать информацию, чтобы ее было легче воспринимать или эффективнее хранить.

Первая форма агрегирования, с которой начинает большинство, — это запрос журналов нескольких процессов. В подразделе «Трассировка» выше говорилось, что OpenTelemetry позволяет извлекать данные, описывающие путь запроса в системе, но это не единственный вариант агрегирования, который может понадобиться. Если у вас есть масштабируемый компонент, то вам может потребоваться отыскать конкретные ошибки во всех репликах.

К счастью, существует масса инструментов, упрощающих группировку журналов из нескольких контейнеров в Kubernetes. Один из самых простых — `ktail` (https://oreil.ly/ktail), который расширяет традиционный интерфейс `kubectl` для объединения журналов по нескольким подам (группам контейнеров). Более сложные инструменты, например Elasticsearch (https://oreil.ly/elasticsearch), создают поисковый индекс для журналов, что позволяет выполнять произвольные запросы к вашим журналам, как при веб-поиске.

Настройка и поддержка индекса журнала — сложная и чреватая ошибками задача, но, к счастью, на данный момент все общедоступные облака и многие стартапы предлагают услугу приема журналов и поиска по ним. Аналогичные услуги доступны для метрик временных рядов. Независимо от того, создаете ли вы систему в общедоступном облаке или в собственной инфраструктуре, использование сервисов для анализа столь важного источника информации, как журналы, часто оказывается правильным выбором.

Помимо сохранения журналов и метрик в системах, специально созданных для эффективного хранения и обработки запросов, иногда требуется сократить объем хранимой информации. Этот шаг можно назвать уменьшением частоты фиксации или агрегированием.

Простейшая форма уменьшения частоты фиксации для метрик — сохранение метрики через бо́льшие интервалы времени. Например, если вы записываете метрику каждую секунду, то через неделю можете записывать ее один раз в минуту. Даже это простое изменение сокращает объем хранимой информации в 60 раз. Конечно, простое уменьшение частоты фиксации может привести к потере информации. Другой вариант — сохранение среднего значения за эти 60 секунд вместо 60 мгновенных значений. Этот простой вид уменьшения частоты фиксации можно использовать для журналов. Лучше всего хранить только определенный класс записей, например ошибки, удаляя при этом менее важную информацию. Также можно просто сохранять лишь часть журналов, хотя это может привести к потере значительных объемов информации. Наконец, можно приложить еще немного усилий и разработать агрегаты для конкретных приложений, которые будут хранить информацию, относящуюся к приложению, и при этом сжимать объем хранимых данных.

Обычно уменьшение частоты фиксации считается хорошим решением, но если вы хотите снизить затраты при сохранении полной точности данных, например, для соответствия требованиям или для безопасности, то можно изменить доступность данных. Выгрузка журналов в «холодное» облачное хранилище может значительно снизить затраты на хранение данных, но одновременно замедлить и усложнить получение информации из этих журналов. Такое решение хорошо подходит для случаев, когда данные используются редко и должны храниться в течение длительного времени. Некоторые решения типа «журналирование как услуга» автоматически реализуют такое многоуровневое распределение данных.

Резюме

Мониторинг является важнейшей частью любой распределенной системы. Он не способствует правильной работе системы, но, когда система работает неправильно, он становится ключевым компонентом, который позволяет понять, что вызвало сбой и как исправить проблему. Изучение правильных подходов к мониторингу и наблюдению за системами имеет решающее значение для быстрого восстановления работоспособности в случае возникновения проблемы или сбоя.

ГЛАВА 15

Использование и обслуживание ИИ

За последние несколько лет ИИ стал ключевой частью многих типов приложений. Нейронные сети и машинное обучение существуют уже несколько десятилетий, но лишь относительно недавно произошел фазовый сдвиг в качестве моделей и приложений ИИ благодаря достижениям в области глубокого обучения и больших языковых моделей (large language model, LLM). Что еще важнее, эти системы захватили воображение разработчиков приложений по всему миру, увидевших самые разные способы применения LLM в своих конкретных бизнес-областях.

ИИ и машинное обучение — сложная тема, на освоение которой могут уйти годы, но, к счастью, благодаря библиотекам и готовым моделям требуется не так много времени, чтобы вы могли начать внедрять ИИ в свое приложение. Эта глава не ставит перед собой цель сделать вас экспертом в сфере ИИ, но в ней вы найдете идеи и подходы к использованию ИИ в вашей системе.

Основы систем ИИ

Прежде чем приступить к рассмотрению деталей использования ИИ в системе, мы должны получить представление об основных идеях, заложенных в ИИ. Большинство людей начинают исследование с модели. Модель — это набор числовых весов, кодирующих знания в нейронной сети. Современные LLM насчитывают триллионы этих весов. В первом приближении модель можно представить как функцию, которая получает набор входных данных и преобразует их в некоторый выходной сигнал.

Однако в отличие от традиционных функций в языках программирования модель нейронной сети проходит процесс обучения, который заключается в тренировке модели на основе большого объема обучающих данных. Обучающие данные содержат пары входных и выходных данных. В ходе обучения входные данные подаются в существующую модель, а ее выходной сигнал сравнивается с ожидаемым выходным сигналом, имеющимся в обучающих данных. Ошибка между ожидаемым и наблюдаемым выходным сигналом возвращается в модель, а та соответствующим образом корректирует веса. Этот итеративный процесс повторяется снова и снова, пока модель не вернет выходной сигнал, совпадающий или почти совпадающий с ожидаемым. Процесс обучения требует больших вычислительных затрат, и для его выполнения используется специализированное оборудование.

Из-за высокой стоимости и сложности обучения LLM многие используют тонкую настройку для дообучения базовых моделей, таких как ChatGPT. Тонкая настройка заключается в том, чтобы дообучить модель общего назначения для лучшего соответствия конкретному применению. В процессе тонкой настройки обычно используются наборы данных меньшего объема и требуется меньше вычислений, поэтому данный подход выглядит эффективнее с точки зрения затрат.

После обучения модели ее можно использовать в приложении. Во время обучения качество модели оценивается по известным входным и выходным данным, а во время использования — по входным данным пользователя, для которых выходные данные неизвестны. Обычно доступ к модели для ее использования обеспечивается через RESTful API, которому можно передавать пользовательские входные данные.

Многие поставщики облачных услуг предоставляют доступ к моделям ИИ в виде услуги. Такие решения могут значительно облегчить первый этап внедрения ИИ в приложение и часто позволяют получить доступ к моделям, которые могут быть недоступны для вашего личного использования. Однако некоторые приложения имеют требования к конфиденциальности данных, а это означает, что модели должны размещаться на наших собственных серверах. О том, как это делается, мы поговорим в следующих разделах.

Для LLM заключительной частью использования является подсказка (prompt). При общении с LLM вам может показаться, что

текст вашего запроса отправляется непосредственно в модель, но в большинстве случаев он упаковывается в более крупный фрагмент текста, который называется подсказкой. Подсказка содержит контекст и инструкции для модели, отсутствующие в вашем конкретном запросе. Например, подсказка может добавить в запрос ваше местоположение или другую известную о вас информацию. Структура подсказки имеет решающее значение для получения хороших ответов от LLM. Акт создания хорошей подсказки известен как проектирование подсказок, или промт-инжиниринг (prompt engineering). В дополнение к приведенным выше примерам промт-инжиниринг можно использовать для добавления дополнительных сведений из традиционных источников данных и для обеспечения соответствия ответов модели требованиям вашего приложения к точности и стабильности.

В этом разделе были описаны компоненты приложения ИИ. В следующих разделах будут представлены более подробные сведения о том, как создать такое приложение.

Размещение модели

Построение модели — большая и сложная задача, описание которой могло бы занять целую книгу (и даже больше), поэтому ее обсуждение выходит за рамки данной главы. Даже если вам хотелось бы обучить или дообучить собственную модель, я рекомендую все же начать с предварительно подготовленных моделей, чтобы вы могли получить представление о том, насколько хорошо ИИ вписывается в ваше приложение. Не всякое приложение идеально подходит для применения текущих моделей ИИ, поэтому, прежде чем тратить время, силы и средства на обучение своей модели, имеет смысл оценить задумку в целом.

Когда вы ищете место для размещения модели, первым делом нужно учесть ее требования и требования вашего приложения к вычислительным ресурсам. Если приложение будет использоваться в интерактивном режиме, то задержка ответов будет иметь решающее значение для создания благоприятного опыта у пользователей. Как правило, пользователи спокойно относятся к задержкам до 250 миллисекунд, а когда задержка достигает нескольких секунд,

начинают отвлекаться и выполнять другие задачи. То есть при использовании модели в интерактивном режиме производительность имеет решающее значение.

Как правило, для быстрой работы моделям LLM требуются высокопроизводительные графические процессоры. Приобретение оборудования с такими графическими процессорами, как и их аренда в облаке, влетит вам в копеечку. Кроме того, учитывая большой интерес к ИИ, сами чипы время от времени может быть трудно достать. Обычно модели LLM не могут работать на мобильных устройствах, таких как телефон или планшет. Если ваше приложение предназначено для использования с мобильных устройств, то вам придется разместить модель в облаке и организовать к ней доступ с мобильных устройств по сети.

Проблема высокой стоимости эксплуатации привела к созданию так называемых малых языковых моделей (small language model, SLM). Эти модели, как и модели семейства Phi от Microsoft, используют гораздо более сфокусированные обучающие данные для достижения разумной производительности при гораздо меньшем количестве весов (миллиарды вместо триллионов). В силу своих характеристик они не могут конкурировать с более крупными моделями, такими как ChatGPT, однако уменьшенный размер позволяет значительно снизить стоимость эксплуатации и задержку в обработке запросов. Для многих приложений такой компромисс оправдан, и малые языковые модели (SLM) вполне могут обеспечивать приемлемое качество при гораздо более низких затратах. Кроме того, поскольку SLM могут работать на мобильном устройстве, они также могут помочь решать задачи пользователям, не желающим делиться своими данными с облачной моделью.

Распространение модели

До сих пор мы в основном говорили об обучении и использовании моделей, но я не упомянул, что в действительности модели сохраняются в файлы и загружаются из файлов для обработки запросов. Существует несколько фреймворков для обучения и использования моделей (мы обсудим их в следующем разделе), и каждый фреймворк имеет собственный формат файла. Например, TensorFlow использует формат

TF2. Если вы не предполагаете распространять свою модель или изменять ее формат, то можете использовать формат, специфичный для фреймворка. Однако в большинстве случаев лучшим выбором является использование универсального формата, не зависящего от фреймворка.

Самый популярный формат для открытых моделей — ONNX (Open Neural Network eXchange), спонсируемый Microsoft, Facebook и Amazon и используемый тысячами разработчиков открытого исходного кода во всем мире. Модели, хранящиеся в формате ONNX, могут обучаться и использоваться для прогнозирования с помощью всех популярных фреймворков и поддерживаются различными «репозиториями» или галереями популярных моделей.

Для многих пользователей, особенно когда они только начинают использовать чужую модель с открытым исходным кодом, лучший способ приступить к работе — загрузить модель из репозитория, подобно тому как пользователи облачных контейнеров загружают образы из репозитория Docker. Репозитории моделей упрощают доступ к самым популярным моделям с открытым исходным кодом. Один из самых популярных таких репозиториев называется Hugging Face. Он включает не только дистрибутивы моделей, но и код, что делает загрузку и начальное использование моделей очень простым делом. В качестве языка программирования почти вся экосистема ИИ использует Python. Чтобы начать работу с моделью из Hugging Face, достаточно написать одну строку кода на Python.

Модели, хранящиеся в этих репозиториях, могут быть довольно большими, поэтому вы вряд ли пожелаете загружать их при каждом запуске приложения (хотя это возможно), так как это может увеличить продолжительность запуска на несколько минут. Вместо этого лучше кэшировать модель локально в файловой системе или в хранилище ключей и значений в памяти. После кэширования модели все экземпляры приложения смогут использовать модель из общего кэша. При перезапуске приложения из-за сбоев или при запуске новой его версии вам не придется ждать, пока загрузится модель.

Последнее соображение, касающееся распространения модели, — безопасное развертывание новых моделей. Обычно модели изменяются реже, чем ваш код, но со временем вы неизбежно будете вносить в них изменения. Как и в случае развертывания кода, крайне важно, чтобы

развертывание модели происходило безопасно. Изменения в модели могут оказать значительное и непредсказуемое влияние на качество и надежность вашего приложения. Чтобы ограничить эти риски, желательно следовать передовым практикам постепенного внедрения и начинать с открытия доступа к новой модели очень небольшому кругу ваших пользователей, например, людям, находящимся в определенном географическом местоположении. Убедившись, что модель работает правильно, вы сможете постепенно открывать доступ к ней все большему количеству пользователей, пока наконец приложение не будет использовать только новую модель.

Разработка с использованием моделей

Сложность эксплуатации моделей заключается в высоких требованиях к вычислительным ресурсам. Независимо от того, размещаете ли вы собственную модель или используете облачную, это означает, что каждый запрос к ней обходится довольно дорого. Если вы не занимаетесь специальными видами разработки, повышающими качество вашей модели (например, промт-инжинирингом), то, вероятно, не имеет смысла оплачивать стоимость полной модели при работе над пользовательским интерфейсом или другими частями сервиса, не связанными напрямую с ИИ.

На этих этапах у вас уже должен быть отлажен доступ к модели, размещенной в облаке Azure или на вашем собственном сервере, через RESTful API. Соответственно, вы можете использовать этот интерфейс, чтобы скрыть детали конкретной модели, используемой для обработки запросов, и тем самым сэкономить расходы, когда вам не нужна полная модель.

При таком подходе вы добавляете параметр в вызов API, который указывает, какую модель использовать. Если вы используете облачный API, то вам может потребоваться разработать обертку для RESTful-сервиса, которая поддерживает этот параметр. При развертывании такого интерфейса совместимости вы выбираете целевую модель. В средах разработки и тестирования можно создать Kubernetes-ресурс `Service`, который ссылается на недорогую модель SLM, а в промышленных окружениях можно использовать полную модель. Уже существует несколько проектов с открытым исходным

кодом, предоставляющих универсальные API доступа к моделям, которые могут охватывать широкий спектр разных моделей.

Аналогично при непрерывной оценке и автоматизированном модульном тестировании приложения имеет смысл развертывать небольшие и недорогие модели, если только вы не собираетесь специально оценивать его качество.

Генерация ответа с дополнением результатами поиска

Пользователи, взаимодействующие с системами ИИ, надеются на их способность понять контекст конкретного запроса, а также учесть самую свежую информацию, доступную в мире. К сожалению, природа моделей ИИ такова, что их уровень знаний фиксируется на моменте времени, когда они обучались. Поскольку модели статичны и основаны на общей информации, генерирование ответов, специфичных для пользователя или отражающих текущее состояние мира, является сложной задачей. Для решения этой проблемы была разработана технология генерации ответа с дополнением результатами поиска (Retrieval-Augmented Generation, RAG).

Системы, поддерживающие технологию RAG, дополняют запрос пользователя традиционными поисковыми запросами, чтобы обеспечить необходимый контекст для получения правильного ответа на вопрос пользователя. Рассмотрим, например, такой запрос.

> *Запрос*: кто из моих друзей ближе всего к моему текущему местонахождению?

Очевидно, что общая модель LLM не имеет ни малейшего представления ни о друзьях человека, задавшего вопрос, ни об их текущем местонахождении. Для решения этой задачи нужно использовать RAG.

При реализации системы с поддержкой RAG в дополнение к запросу пользователя система выполняет еще два независимых запроса. Первым может быть традиционный запрос к базе данных для поиска по списку контактов пользователя, чтобы определить круг его друзей. Второй запрос может быть к службам определения местоположения телефона для выяснения текущего местонахождения пользователя.

Результаты этих двух традиционных запросов объединяются с запросом пользователя и затем отправляются модели. Пример запроса в системе с поддержкой RAG может выглядеть так.

> *Запрос*: кто из моих друзей с адресом `${address}` и текущим местонахождением `${location}` ближе всего к моему текущему местонахождению?

Здесь видно, как персонализированная и актуальная информация, возвращаемая поисковыми запросами RAG, встраивается в пользовательский запрос, чтобы дать модели LLM достаточно информации для правильного ответа.

RAG — отличная технология для опытных разработчиков, у которых может не быть большого опыта в области ИИ для создания сложных приложений ИИ. Объединяя традиционные запросы к базе данных с запросами пользователей на естественном языке, можно создавать успешные языковые интерфейсы, кажущиеся очень персонализированными. Еще одно преимущество RAG — конфиденциальность пользователя. Никакая личная информация пользователя не передается модели LLM, где она может быть случайно раскрыта другим пользователям, — вся информация доступна только пользователю, имеющему к ней доступ, и передается модели в виде расширенной подсказки. Благодаря своей объяснимости и функциям защиты конфиденциальности RAG является одной из самых популярных методик для современных интерактивных систем ИИ.

Тестирование и развертывание

В настоящее время необходимость тестирования хорошо понимается и принимается как обязательная практика всеми, кто хочет иметь надежный и безотказный сервис. Но что означает модульное тестирование для приложений ИИ, когда внесение изменений в подсказку может оказать существенное влияние на ответы, получаемые от модели?

Первый важный сдвиг в мышлении, необходимый для успешного тестирования приложений ИИ, — переход от тестирования корректности к статистическому тестированию уровня качества. Нас учили при разработке модульных тестов оценивать корректность результата,

возвращаемого функцией, его совпадение с нашими ожиданиями. И действительно, существуют целые фреймворки, основанные на выражении ожидания, что `Value(A).equals(B)`. Такие тесты непригодны для приложений ИИ.

В своих тестах вместо поиска точных совпадений мы оцениваем, является ли ответ «достаточно хорошим». Переключив внимание с понятия «корректный» на понятие «достаточно хороший», мы понимаем, что для оценки недостаточно ответа на один запрос. По мере изменения модели ответ на любой запрос может кардинально меняться. Фактически изменение модели может кардинально улучшить ответ на один тип запросов и кардинально ухудшить ответ на многие другие типы.

Таким образом, чтобы правильно протестировать приложения ИИ, нужно рассматривать качество их работы на большом наборе подсказок и оценить, улучшает изменение качество ответов или ухудшает. Если изменение оказывает нулевое или положительное влияние на общее качество ответов, то считается, что такое изменение успешно проходит тест. Если качество снижается, то тест считается непройденным.

Чтобы на самом деле реализовать такой тест, нужен способ оценки качества ответа. Эта задача сама по себе может быть очень сложной. Очевидно, что было бы слишком дорого нанимать людей для оценки качества каждого ответа — нужно какое-то автоматизированное решение. К счастью, для частичной оценки можно использовать сами модели LLM. Модель можно применять не только для получения ответа на вопрос пользователя, вы также можете спросить ее — является ли ответ, который она дала, высококачественным. Подсказка для такого случая выглядит примерно так.

> *Подсказка*: на вопрос `${input}` был дан ответ: `${response}`. Является ли такой ответ высококачественным?

Может показаться, что такой подход — все равно что дать заключенным ключи от камеры, но на самом деле он во многих случаях дает неплохие результаты.

В дополнение к такому тестированию качества на основе ИИ, прежде чем выпустить обновления в эксплуатацию, крайне важно предусмотреть для пользователей возможность оставлять свои отзывы. Если

в других системах можно искать ошибки и выявлять задержки, чтобы определить, безопасно ли продолжать развертывание, то в приложениях ИИ следует также проверять и оценивать удовлетворенность пользователей полученными ответами.

ИИ меняет способ создания приложений и, соответственно, изменяет способ их тестирования. Однако это не снижает важности тестирования и оценки каждого вносимого изменения, чтобы убедиться, что оно не влияет на общее качество приложения.

Резюме

Всего за несколько лет искусственный интеллект изменил наше представление о том, какими могут быть наши приложения и ожидания пользователей относительно способов взаимодействия с этими приложениями. Однако обработка запросов с помощью ИИ — это всего лишь еще одна функция, которая предоставляется распределенной системой нашим пользователям. В этой главе было дано введение в основные концепции и методы создания приложений с искусственным интеллектом, которые помогут любому, кто знаком с распределенными системами, дополнить свои приложения поддержкой ИИ.

ГЛАВА 16

Распространенные паттерны отказов

До сих пор в книге описывались различные паттерны проектирования, помогающие создавать распределенные системы. Эта глава отличается от остальной части книги — вместо того чтобы помочь вам понять, что и как делать, она рассказывает, чего делать *не надо*. В практике разработки, эксплуатации и отладки систем определенные виды проблем повторяются снова и снова. Представленные здесь паттерны делятся на две категории: ошибки, которые допускаются при создании систем, и часто встречающиеся виды выхода систем из строя. Понимая, чего не следует делать и что нужно попытаться предотвратить, мы можем извлечь уроки из этих типичных ошибок, чтобы не повторить их в будущем.

Грохочущее стадо

«Грохочущее стадо» получило свое название в результате сравнения с бегущим стадом бизонов или других крупных животных. По отдельности эти животные могут быть управляемыми, но, когда бегут вместе, они способны уничтожить все, что встречается на их пути. Проще всего понять, что такое «грохочущее стадо», — представить, что вы взаимодействуете с веб-сайтом, который ведет себя неправильно. Вы пытаетесь перейти в определенное место, индикатор загрузки медленно вращается, но видимого прогресса не наблюдается, и в какой-то момент терпение заканчивается и вы нажимаете кнопку перезагрузки. Вы можете этого не знать, но в этот момент вы стали «грохочущим стадом».

У любого конкретного приложения есть максимальная емкость. Обычно мы пытаемся установить емкость наших приложений так,

чтобы она была больше любой нагрузки даже при пиковом наплыве пользователей. К сожалению, иногда из-за сбоев, потери емкости или непредвиденного всплеска интереса со стороны пользователей количество отправленных сервису запросов превышает его возможности. Допустим, что перегрузку вызывает трафик, насчитывающий X запросов в секунду. Перегрузка — это плохо; обработка многих, если не всех, из этих X запросов в секунду прерывается по тайм-ауту, но то, что происходит дальше, еще хуже. В следующую минуту те же самые X запросов превращаются в $2X$ запросов. Это удвоение является результатом сложения нового трафика (X), а также повторного запуска всех предыдущих запросов. Результат — еще больше сбоев, еще больше повторов и еще больше новых запросов. Так плохая ситуация превращается в непоправимую. Это особенно верно, когда задействовано несколько цепочек вызовов и несколько алгоритмов повтора. Превращение одного запроса в множество запросов может быстро привести к перегрузке системы.

Создавая системы, мы встраиваем повторные попытки, стремясь обеспечить безотказную обработку запросов, но, слепо повторяя попытки в случае ошибок, мы увеличиваем численность «грохочущего стада», ухудшая ситуацию. Первый способ исправить ситуацию на стороне клиента — использовать экспоненциальную задержку. Вместо немедленного запуска повторной попытки клиент некоторое время ждет и только потом пробует отправить запрос еще раз. Каждый раз, когда клиент видит ошибку, он удваивает время ожидания.

Экспоненциальная задержка — хороший вариант, но несовершенный. В дополнение к экспоненциальной задержке также полезно добавлять небольшую *флуктуацию*. Флуктуация — это доля случайности, которая распределяет нагрузку по времени. Без этого может возникнуть ситуация, когда все одновременно прерывают выполнение запросов, ждут одинаковое время и снова одновременно вызывают сервис. Флуктуация особенно важна в ситуациях, когда одни программы взаимодействуют с другими. Люди достаточно случайны, чтобы добавлять флуктуацию, обусловленную особенностями их реакции, но машины — нет.

Последний и более полный способ решения проблемы «грохочущего стада» — добавить на стороне клиента *автоматический выключатель*, который срабатывает, когда частота ошибок превышает определенный порог, и останавливает весь трафик на некоторое

время, чтобы система могла восстановиться. Наконец, чтобы помочь в восстановлении, иногда желательно постепенно возвращать трафик в систему. Многие системы могут стабильно работать при высокой нагрузке, но их необходимо доводить до этого уровня постепенно. Одновременная отправка всего трафика на сервер может снова привести к проблеме «грохочущего стада».

Отсутствие ошибок — это ошибка

Теперь перейдем от проблем, обусловленных большим трафиком, к проблемам, которые характеризуются слишком малым трафиком. Обычно в системы добавляют средства мониторинга и оповещения, которые срабатывают, когда в системе возникает слишком много ошибок. Очевидно, что слишком много ошибок — это проблема, которая требует внимания со стороны человека, чтобы тот заглянул в систему, определил причину и восстановил нормальную работу. А как насчет слишком малого количества ошибок? Это может показаться нелогичным, но некоторое количество ошибок является устойчивым состоянием для большинства распределенных систем. Обычно очень сложно, да и не нужно пытаться предотвратить появление любых ошибок, поэтому всегда существует некоторое количество ошибок, обусловленных внешними причинами, которые обрабатываются повторными попытками и другими алгоритмами исправления ошибок. Следовательно, *полное отсутствие ошибок*, скорее всего, указывает на серьезную проблему, а не на то, что все идет отлично.

Чтобы понять, как такое может произойти, представьте мониторинг системы, которая добавляет субтитры к фильмам, загруженным в хранилище. Всякий раз, когда загружается фильм, система преобразует диалоги в текст, добавляет этот текст в фильм в виде субтитров и сохраняет обновленный фильм обратно в хранилище. Вообразите, что произойдет, если процесс, загружающий фильмы, случайно потеряет разрешение на загрузку этих фильмов. Если мониторинг отслеживает только условия, когда количество ошибок превышает некоторый порог (скажем, 10 %), то он никогда не выдаст оповещение об этом состоянии. Если загрузчик не сможет загрузить фильмы, то фильмы перестанут обрабатываться и никаких ошибок не возникнет, соответственно уровень ошибок будет равен 0 %. Но совершенно очевидно, что если загрузчик не может загрузить фильмы, то это означает, что

система потеряла работоспособность. Оповещение как о слишком большом, так и о слишком малом количестве ошибок (или о слишком малом количестве запросов в целом) позволит вам обнаруживать (и исправлять) такие проблемы.

Клиентские и ожидаемые ошибки

Следующий паттерн тоже связан с ошибками в системе, но на этот раз с ошибками, игнорирование которых часто считается нормальной реакцией. Многие системы принимают пользовательский ввод и запросы на обработку. В таких системах пользователи (или клиенты в целом) могут отправлять запросы, содержащие ошибки и непригодные к обработке в текущем виде. При оценке надежности сервиса, очевидно, не следует рассматривать ошибки, обусловленные плохими клиентскими запросами, как истинные ошибки, поскольку сервис не контролирует то, что вводят клиенты. Однако на практике такие ситуации, которые можно характеризовать как «не мои проблемы», часто приводят к игнорированию истинных системных ошибок. Представьте, например, ошибку авторизации. Конечно, отправка запроса неавторизованным пользователем является ошибкой клиента и не указывает на проблему с надежностью. Но что, если внезапно *все* пользователи окажутся неавторизованными? Это может указывать на проблему в нашей подсистеме авторизации. И если относиться ко всем ошибкам несанкционированного доступа как к «ошибкам пользователей», не пытаясь увидеть аномалии, как, например, когда все пользователи оказались неавторизованными, то, вероятно, вы не заметите, когда подсистема авторизации выйдет из строя.

То же относится к ожидаемым ошибкам. В разделе «Отсутствие ошибок — это ошибка» выше я упоминал, что существует определенный ожидаемый уровень ошибок в системе, но это не относится к синтетическим запросам, которые создает сама система. При мониторинге системы важно отделять реальные клиентские запросы, для которых ожидается некоторый небольшой процент ошибок, и *синтетические*, или сгенерированные системой, запросы, которые целиком и полностью контролируются вами. В случае с синтетическими запросами вы контролируете как клиентскую, так и серверную сторону взаимодействия, и, следовательно, при обработке таких запросов не должно

возникать никаких клиентских или ожидаемых ошибок. Объединив мониторинг реального трафика с мониторингом синтетических запросов, можно быстро выявлять ситуации, когда изменения в вашем коде порождают ошибки, которые в ином случае были бы проигнорированы как ошибки клиента.

Ошибки управления версиями

В любое программное обеспечение со временем вносятся некоторые изменения, но клиенты требуют и ожидают, что они смогут продолжить взаимодействовать с системой привычным им способом. Если вам вдруг потребуется обновить каждого клиента одновременно с системой, то это означает, что на самом деле вы создали несогласованную распределенную систему. Любая система состоит из внешнего API, который предоставляется клиентам, и внутреннего представления этого API в памяти. На первый взгляд кажется, что лучшим решением будет сделать эти два представления одинаковыми. Если они разные, то придется добавить логику преобразования, чтобы совместить внешнее представление с внутренним и наоборот. И конечно, в самом начале внешнее и внутреннее представления идентичны.

На практике разделение внутреннего представления API на диске или в памяти и внешнего, обслуживающего клиентов, дает огромные выгоды. В первую очередь это позволяет иметь несколько версий внешнего клиентского API, поддерживаемых одной и той же внутренней версией в памяти. Благодаря этому разные клиенты смогут взаимодействовать с системой, используя разные версии ваших API, и вам не придется жестко синхронизировать клиентов с приложением. Кроме того, наличие отдельного внутреннего представления упростит дальнейшее развитие API и вы сможете добавлять, удалять или объединять поля по мере необходимости и без ведома клиентов при условии, что поддерживаете логику преобразования между внешним и внутренним представлениями.

Преимущества такого разделения особенно часто заметны при поддержке нескольких клиентов, но то же самое верно в отношении внутреннего хранилища. Как бы то ни было, вашему приложению определенно потребуется хранить объекты, созданные в API, в некотором

хранилище данных. И система должна поддерживать возможность работы с разными версиями хранилища. Если версия хранилища будет тесно связана с определенной версией вашего кода, то это может сильно затруднить развертывание изменений в слое хранения или откат к предыдущей версии при появлении ошибок.

На первый взгляд все это может показаться слишком сложным, однако создание приложения, поддерживающего разные версии API — внешнего (клиентского), внутреннего (в памяти) и доступа к хранилищу (на диске или в базе данных), — а также имеющего логику преобразования между ними, даст значительные преимущества в плане гибкости, что в долгосрочной перспективе принесет пользу вашему проекту.

Миф о необязательных компонентах

Распределенные системы — это практически живые организмы. На начальном этапе мы проектируем их одним образом, а затем, в ответ на сбои, проблемы масштабирования и изменения в бизнесе, трансформируем и адаптируем в соответствии с новыми требованиями. Одним из наиболее распространенных изменений является добавление кэшей в различных местах для повышения производительности и снижения нагрузки.

Обычно, когда кэши добавляются в систему, они рассматриваются как необязательные компоненты. В конце концов, кэш дает всего лишь улучшение производительности. При необходимости система сможет получить данные из первичного источника. Проблема с теоретически необязательными компонентами заключается в том, что со временем они становятся обязательными. Рассмотрите возможность добавления кэша в систему. Это необязательное улучшение производительности при первоначальном добавлении. Однако со временем, по мере увеличения нагрузки на систему, растет и ее производительность. В результате система становится все более зависимой от наличия кэша, пока наконец не окажется в таком состоянии, что уже не сможет обеспечить нужную производительность без кэша.

К сожалению, эта возросшая зависимость часто остается незамеченной, и общая надежность кэшей в системе не поддерживается на том

уровне, какой должен обеспечиваться в соответствии с их возросшей критичностью для общей стабильности системы.

Одним из наиболее типичных проявлений такой незаметности зависимости является разница между локальным и глобальным кэшем. Локальный кэш, как следует из названия, — это кэш, который поддерживается локально в памяти приложения, часто в форме простой хеш-таблицы. Такие кэши легко реализуются. К сожалению, с ростом системы пропорционально растет и общий объем таких локальных кэшей, и в конечном счете требования системы к памяти становятся слишком большими. Существует также тенденция добавлять много таких кэшей в рамках одного приложения, потому что это так легко сделать, а вот визуализировать и контролировать использование памяти такими кэшами сложно.

Кроме того, локальные кэши усложняют оптимизацию потребления памяти. Следовательно, если вдруг вы поймаете себя на том, что добавляете в систему локальный «необязательный» кэш или любой другой теоретически «необязательный» компонент, то имейте в виду, что фактически вы вносите постоянное (и обязательное) изменение, которое заслуживает такого же пристального внимания, как любое другое значительное изменение.

Ой, мы все стерли

К сожалению, в любой системе случаются ошибки. Ошибки и сбои неизбежны. Поэтому важно учитывать, как ошибки и сбои могут повлиять на общее состояние создаваемой нами системы. При этом важно думать не только о работе отдельных частей, составляющих систему, но и о взаимодействиях между ними. Неспособность предвидеть, как части системы могут взаимодействовать друг с другом в случае сбоя, часто оказывается причиной серьезных проблем. Одна из них — неконтролируемое удаление данных из-за какого-то непредвиденного эффекта взаимодействия.

Удаление всех данных в вашей системе выглядит таким экстремальным провалом, что порой трудно осознать возможность этого из-за незначительной ошибки. По этой причине полезно изучить примеры подобных происшествий, случившихся в реальности.

Рассмотрим пример с системой хранения фотографий. Представьте, что она состоит из четырех компонентов, таких как:

- база данных с информацией о пользователях;
- сервер RESTful API, реализующий пользовательский API;
- система хранения файлов с фотографиями;
- система сбора мусора, которая удаляет фотографии пользователей после удаления их учетных записей.

Когда все работает штатно, пользователь инициирует удаление учетной записи через сайт, после чего информация о нем удаляется из базы данных. Далее система сбора мусора проверяет каждую фотографию, хранимую в файловой системе, отыскивая соответствующего пользователя в базе данных. Если поиск закончился неудачей, то фотография удаляется. Цель системы сбора мусора — обеспечивать соответствие законам о конфиденциальности данных, а также предотвращать утечку фотографий и препятствовать бесконечному росту файловой системы.

Теперь рассмотрим фрагмент кода в RESTful API, выполняющий поиск пользователя. Он инициирует соединение с базой данных, ищет информацию о пользователе и возвращает ее вызывающей стороне. Но что, если по какой-то причине база данных окажется недоступна? Предположим, что человек, писавший этот код, решил вернуть код 404 (Not found — «не найдено»), указывающий, что пользователь не найден. Возможно, это не лучший способ сообщить о неудаче — код 500 (Internal error — «внутренняя ошибка») определенно лучше, но пока вы не подумаете о системе сбора мусора, вам будет трудно понять, насколько важен этот код ошибки. Когда система сбора мусора будет запрашивать информацию о пользователях, то для нее *все* пользователи будут выглядеть отсутствующими и она удалит *все* фотографии. К счастью, инженеры предвидели катастрофические проблемы и регулярно создавали резервные копии, так что они восстановили фотографии из резервных копий; но сервис был недоступен в течение многих часов, пока производилось восстановление из резервных копий в «холодном» хранилище. В этот период доверие пользователей к надежности хранения чего-то столь глубоко личного и важного, как семейные фотографии, было поколеблено, и сервису пришлось немало потрудиться, чтобы восстановить доверие.

Так что же пошло не так? В этой системе есть вторичные базовые точки отказа, которые вызвали катастрофический сбой. Первая — человеческий фактор. Человеку сложно постоянно держать в голове всю информацию о распределенной системе. Особенно если разработкой пользовательского API занимается одна команда, а разработкой системы сбора мусора — другая, возможно находящаяся в другой точке мира. Возникает соблазн сосредоточиться на этой человеческой ошибке, потому что это разработчик допустил ошибку и «вызвал» сбой. К сожалению, человеческие ошибки неизбежны при создании распределенных систем. Их нельзя предотвратить, можно только ослабить их последствия. Люди будут совершать ошибки независимо от возраста и опыта, и наши системы должны предвидеть и уменьшать последствия этих ошибок.

Это рассуждение приводит нас ко второй точке отказа: к процессу или техническому сбою в системе, а именно к ограничению скорости и автоматическому отключению в логике сбора мусора. В системе при нормальной работе учетные записи удаляют всегда примерно одинаковое количество пользователей. Допустим, один процент всех пользователей удаляет свои учетные записи в определенный день; это приведет к относительно постоянной скорости удаления изображений. Когда произошла разбираемая нами ошибка, скорость удаления фотографий увеличилась в 100 раз. Это явно необычно. Система должна была заметить такую высокую скорость удаления и прервать работу сборщика мусора. Первым усовершенствованием системы было бы добавление настроек, регулирующих частоту выполнения операции удаления. С помощью таких настроек можно было бы установить верхнюю границу удалений в секунду, чтобы значительно замедлить скорость удаления всех фотографий и дать инженерам время среагировать и остановить процесс. Второе усовершенствование — «автоматический выключатель», очень похожий на автоматические выключатели в наших домах: когда высокая частота удалений сохраняется в течение длительного времени, система сбора мусора должна прекратить удаление, пока человек не сбросит выключатель. Конечно, эти технические усовершенствования не могли полностью предотвратить последствия человеческой ошибки, но они значительно уменьшили бы негативное влияние этих последствий на клиентов сервиса и на бренд самого сервиса.

Такие случайные ситуации «мы все удалили» на удивление широко распространены, потому что как разработчики мы склонны фокусироваться на «счастливом пути», когда все работает правильно. В этом мире вызов пользовательского API никогда не может вернуть неправильный ответ. Однако опыт распределенных систем учит нас, что мы всегда должны думать о мире, в котором все идет правильно, и о том, что может произойти, когда что-то пойдет не так.

Проблемы с широтой входных данных

Один из самых удручающих видов ошибок — ошибки, просочившиеся через все ваши тесты и остающиеся незамеченными, пока их не обнаружит клиент. Такие ошибки расстраивают клиентов. Они задают вопрос: «Разве вы не тестируете свой код?» — на который трудно что-то ответить в таких случаях. Эти ошибки смущают еще и тем, что зачастую долгое время остаются незамеченными в вашем коде, пока кто-то не столкнется с ними и не заявит об этом во всеуслышанье.

Иногда эти ошибки остаются незамеченными, потому что возникают только при очень редком стечении обстоятельств, например при сложном условии гонки, которое случается раз в месяц. Такие ошибки на самом деле гораздо реже расстраивают конечного пользователя, потому что выглядят как простая ненадежность. Нас всех приучили решать проблемы методом «перезагрузи и попробуй снова», что обычно исправляет такие ошибки.

Намного хуже ошибки, которые возникают в 100 % случаев, но только у небольшого числа пользователей. Я называю эти проблемы проблемами компилятора, потому что они обычно возникают, когда набор допустимых входных данных в системе очень широк. В такой системе определенные комбинации входных данных могут вызывать сбой в 100 % случаев, но только определенные пользователи вводят такие данные.

Проблемами компилятора они называются потому, что компилятор был одной из первых программ, в которых проявились такие проблемы. Пространство допустимых программ на любом языке программирования фактически бесконечно. При наличии достаточного

количества программистов, работающих над достаточно большим количеством программ, рано или поздно один из них напишет программу, которая по всем правилам является допустимой, но из-за ошибки в компиляторе терпит сбой. Иногда такие входные данные являются продуктом случайности и человеческой природы, но чаще они создаются злонамеренными фаззинг-программами, которые посылают в API случайные данные, стараясь вызвать сбой или как-то иначе воспользоваться слабостью в API. Хорошо и печально известной версией таких атак были атаки с использованием SQL-инъекций, типичные для раннего Интернета, когда фрагменты SQL-кода (например, `'; DROP TABLES'`) передавались во входных данных там, где ни один разработчик не мог ожидать получить SQL-выражение (например, в поле ввода номера кредитной карты). К настоящему времени атаки с использованием SQL-инъекций остались в прошлом, но попытки взлома легитимных пользователей или атаки методом фаззинга по-прежнему довольно распространены.

Так или иначе, эта ошибка является результатом недостаточного охвата тестирования. Если программа протестирована целиком и полностью, то такие сбои невозможны, но пространство допустимых входных данных для многих API бесконечно или почти бесконечно. Как гарантировать полный охват тестированием в такой ситуации? Есть два основных решения этой задачи.

Первое — записать репрезентативную выборку реальных входных комбинаций. Конечно, никакая выборка не может быть исчерпывающей, но если записать (например) все комбинации, введенные пользователями в последние два месяца, то можно быть достаточно уверенными, что добропорядочный пользователь вряд ли вызовет сбой. Однако, составляя такие выборки, нужно уделить особое внимание месту и времени сбора выборки. Если сервис оказывает услуги по всему миру, а вы записываете входные комбинации данных только в Северной Америке, то, скорее всего, вы пропустите много ошибок, связанных с азиатскими символами. Аналогично если днем и ночью вводятся разные данные, а вы записываете их в выборку только в определенное время, то есть риск пропустить ошибки, связанные с временем суток. Качественный охват тестированием достигается, только когда тестовые данные являются репрезентативными для *всех* пользователей.

Второе решение — использовать для тестирования рандомизированные данные. Такие тесты могут быть значительно более всеобъемлющими, но и создавать их гораздо сложнее. Для тестирования простого API, который принимает лишь несколько целых чисел, легко сгенерировать случайные числа в тестах, но для более сложных API с богатой семантикой может потребоваться написать собственную логику генерации случайных входных данных (например, случайных, но допустимых IP-адресов). Последние достижения в области LLM и генеративного ИИ позволяют генерировать допустимые, но случайные входные данные для еще более широкого круга классов данных.

Сочетание тестирования с использованием ранее вводившихся, а также рандомизированных входных данных может помочь гарантировать надежную работу нового кода с существующими пользователями и его защищенность от возможных атак с использованием случайных или вредоносных входных данных.

Обработка устаревших заданий

Мы все сталкивались с неотзывчивыми системами, например с системой электронной почты, в которой не работает поиск, или с интернет-магазином, не показывающим следующую страницу. В этой главе приводится много примеров неправильной работы систем. Что мы все делаем, столкнувшись с такими системами? Правильно, нажимаем кнопку *перезагрузки* и тем самым создаем для наших сервисов дополнительный способ отказа.

В разделе «Грохочущее стадо» в начале этой главы вы узнали, как такие повторные попытки могут быстро вывести из строя всю систему. Проблема, обсуждаемая в этом разделе, имеет несколько иной характер — когда система пытается наверстать упущенное.

Представьте систему, которая ищет фотографии, похожие на фотографию с нашим любимым питомцем. Запросы в этой системе принимаются и обрабатываются асинхронно, и после обработки результаты возвращаются отправителю запроса. Но что, если к моменту возврата результатов отправитель запроса уже не ждет их? Что, если он нажал кнопку *«перезагрузить»* и повторно отправил один или даже множество дополнительных запросов? Поскольку никто не ждет

результатов, работа, проделанная для их получения, оказывается проделанной впустую и, что еще хуже, из-за потери времени на получение этих результатов задерживаются ответы на запросы других пользователей.

Чтобы понять, как это происходит, представьте, что в нашей системе обработки фотографий произошел сбой в микросервисе распознавания объектов. Обработка запросов останавливается, но новые запросы продолжают поступать. Это похоже на затор на сборочном конвейере: даже если какой-то станок перестал обрабатывать детали, новые заготовки продолжают поступать по конвейерной ленте. Эта проблема была быстро обнаружена и устранена, и система продолжила обработку фотографий. Но что происходит дальше?

За то время, пока система не выполняла никакой работы, на входе вышедшего из строя компонента накопилось множество необработанных заданий. Если система близка к оптимальной нагрузке и находится в устойчивом состоянии, что желательно для максимальной эффективности, то ей будет очень трудно наверстать это отставание. Пока существует отставание, новые запросы обрабатываются с задержкой. Нередко в таких ситуациях короткий сбой, скажем менее 10 минут, может вызвать увеличение задержки обработки запросов, ощущаемой в течение многих часов, и соответствующее ухудшение качества обслуживания клиентов.

А можно ли что-то предпринять, чтобы наши системы лучше реагировали на такие задержки и увеличение задержек? Да, можно. Существует три подхода к решению этой проблемы, которые можно использовать по отдельности или вместе. Первый — добавление тайм-аутов, ограничение времени действия запросов. Каждый запрос, поступающий в систему, должен быть обработан в течение некоторого времени, по истечении которого его следует объявить недействительным и невыполненным. Когда время обслуживания запроса истекает, его обработка прерывается. Тайм-ауты могут помочь системе отбросить устаревшие задания при значительной перегрузке. Однако важно отметить, что сами по себе тайм-ауты не решают описанную выше проблему. Если задержка обработки каждого запроса высока сама по себе, пусть и немного ниже значения тайм-аута, ваша система будет казаться вялой, но она будет отбрасывать задания, чтобы вернуться в нормальное состояние.

Второе решение — автоматическое масштабирование при изменении времени задержки. Мы привыкли к автоматическому масштабированию при изменении нагрузки на процессор или потребления памяти, но часто более разумно использовать автоматическое масштабирование на основе метрик, описывающих характер обслуживания пользователей, таких как задержка обработки запросов. Если система сможет автоматически масштабироваться при изменении времени задержки запроса, то это может помочь ей обработать задания, накопившиеся за время простоя. Даже простое ручное масштабирование после сбоя может стать эффективным средством смягчения последствий.

Последний подход — явное увеличение приоритета новых запросов по сравнению со старыми, когда очереди становятся слишком длинными или задержка оказывается слишком высокой. Этот подход являет собой форму сортировки, приносящей в жертву опыт отдельных пользователей, которые, по всей вероятности, уже не ждут ответов на свои запросы, чтобы обработать более поздние запросы, возможно от тех же самых пользователей, которые нажали кнопку перезагрузки.

Проектирование надежных распределенных систем сопряжено не только с предотвращением сбоев, но и с быстрым восстановлением работоспособности после сбоев, когда те все же происходят. Пользователи гораздо терпимее относятся к коротким сбоям, когда система быстро восстанавливается, чем к длительным, когда восстановление после «исправления» проблемы занимает много времени.

Проблема второй системы

До сих пор мы рассматривали разные типичные проблемы, возникающие в распределенных системах. Прочитав эту главу, вы, возможно, узнали в описании некоторые проблемы, присутствующие в вашей текущей системе. Видя потенциальные возможности появления сбоев, вы хотели улучшить систему, но как это сделать?

Иногда достаточно нескольких простых изменений и совершенно понятно, как внести эти изменения, после чего останется только развернуть их и покончить с этим. К сожалению, чаще проблемы выглядят настолько большими и настолько сложными, что наиболее

эффективным решением кажется создание новой системы с нуля, которая заменит существующую систему. Такой подход привлекателен по ряду причин: построить с нуля проще, чем вносить изменения; гораздо интереснее создавать что-то новое, чем возиться с исправлением существующего решения; и, наконец, создание новой системы при наличии старой позволяет без спешки отладить новую систему и передать ее в промышленную эксплуатацию, только когда она будет полностью готова.

Привлекательность всех этих предполагаемых преимуществ превращает создание второй системы в одну из самых распространенных неудач. Именно эта последняя фраза, «только когда она будет полностью готова», оказывается проклятием второй системы. На практике многие вторые системы никогда не становятся готовыми. Существует множество организационных причин, почему это может произойти, но наиболее распространенная — вторая система вечно догоняет текущую и не получает опыта промышленного использования, необходимого для замены исходной системы.

Вторая система постоянно играет роль догоняющей именно потому, что не является текущей. Это означает, что, пока она создается, все важные функции, исправления ошибок и улучшения, востребованные бизнесом, попадают в исходную систему. Предположим, что на завершение второй системы уходит шесть месяцев. К моменту ее завершения в оригинальной системе появились новые функции и исправления, которые необходимо реализовать в новой системе, чтобы она догнала старую. Если вы не готовы значительно увеличить штат команды, создающей вторую систему, то так и будете играть в догонялки.

Необходимость в увеличении штата для разработки второй системы влечет за собой другой недостаток, а именно неготовность второй системы к эксплуатации. Чтобы стать готовой, любой системе нужен период опытной эксплуатации. Каким бы хорошим вы ни считали свой код, я гарантирую, что в нем имеются ошибки, которые будут обнаружены только после выпуска в эксплуатацию. Но когда ошибки обнаруживаются во второй системе, ее стремятся отключить, потому что в оригинальной системе этих ошибок нет. Это резко снижает возможности совершенствования второй системы, чтобы сделать ее действительно готовой к эксплуатации. Подобно накоплению новых функций, которые появились, пока создавалась вторая система,

накапливается и долг по готовности к эксплуатации, и второй системе очень трудно наверстать упущенное.

Что можно сделать, учитывая, что в вашей распределенной системе наверняка имеются конструктивные проблемы или функции, которые нужно написать или исправить, а реализация второй системы часто обречена на провал? Решение кроется в природе самой распределенной системы. Вместо замены всей распределенной системы сосредоточьтесь на совершенствовании отдельных микросервисов. Это поможет продолжить движение вперед и избежать затрат на поддержку двух систем сразу. Аналогично имеет смысл заняться абстракциями, отделяющими основную бизнес-логику от компонентов, которые может потребоваться заменить с развитием системы, таких как слой хранения. При таком разделении вы сможете развивать служебные компоненты системы, такие как хранилище, независимо от бизнес-логики и даже использовать два разных слоя хранения для оценки и тестирования.

Как и во многих других аспектах нашей жизни, идея выбросить старое и построить что-то новое выглядит заманчивой, но в итоге новые системы оказываются хуже существующих систем после их адаптации и улучшения с точки зрения удовлетворения новых потребностей бизнеса и пользователей.

Резюме

Эта глава не похожа ни на одну из предыдущих. Вместо обсуждения, как строить распределенные системы, она рассказывает, как *не надо* их строить. Надеюсь, что обзор распространенных видов сбоев распределенных систем поможет вам отыскать проблемы в ваших существующих системах, а также создавать более надежные системы в будущем. Изучение чужих ошибок предохранит вас от их совершения в своих системах.

Заключение

Все компании, независимо от происхождения, становятся цифровыми. Такие преобразования требуют создания API и сервисов, используемых в мобильных приложениях, в устройствах, подключенных к Интернету вещей (IoT), и даже в беспилотных транспортных средствах 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 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 технологической 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, упрощающими и ускоряющими развертывание 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.

Паттерны Ambassador, Sidecar, а также шардирование, FaaS, очереди задач и многое другое 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) — корпоративный вице-президент в Microsoft, где отвечает за управление и руководство такими направлениями, как Azure, Azure Arc, Kubernetes на Azure, Linux на Azure и PowerShell. До Microsoft он работал в Google, в подразделении Google Cloud Platform, где стал одним из основателей проекта Kubernetes и помогал в создании таких API, как Deployment Manager и Cloud 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 Сандерс (Robin Sanders), двумя детьми и кошкой Миссис Железная Лапа.

Иллюстрация на обложке

Животное на обложке книги — рисовка, или яванский воробей *(Padda oryzivora)*. Эту птицу ненавидят в дикой природе, но обожают в неволе. Фермеры убивают тысячи диких рисовок, защищая свой урожай. Они также ловят этих птиц для продажи и употребления в пищу. Несмотря на неравную борьбу, рисовки продолжают процветать в Индонезии, на Яве и Бали, а также в Австралии, Мексике и Северной Америке.

Оперение верхней части тела и груди рисовки голубовато-серое. Макушка и горло черные. У этой птицы большие белые пятна на щеках и очень сильный клюв. Верхняя и нижняя стороны клюва ярко-красные, в то время как границы клюва более светлого цвета, а его вершина белесая. Глаза коричневые, окружены светло-красным кольцом. Песнь рисовки начинается с отдельных нот, похожих на звон колокольчика. Потом она перерастает в сплошную трель со звуками трещотки, высокими и низкими нотами.

Основную часть рациона птиц составляет рис, но они не брезгуют и мелкими семечками, травинками, насекомыми и цветковыми растениями. В дикой природе рисовки 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, причем большинство птиц откладывают яйца в апреле-мае.

Рисовка занесена в Красную книгу Международного союза охраны природы (МСОП) из-за продолжающегося сокращения ее естественной среды обитания и массового уничтожения и отлова. Многие из животных, изображенных на обложках книг издательства O'Reilly, находятся под угрозой исчезновения. Все они важны для нашей природы.

Иллюстрация на обложке, созданная Карен Монтгомери (Karen Montgomery), основана на черно-белой гравюре из книги *The Royal Natural History* («Королевская естественная история») Ричарда Лидеккера (Richard Lydekker).

Брендан Бёрнс

Распределенные системы. Паттерны и парадигмы для масштабируемых и надежных систем на основе Kubernetes

2-е издание

Перевел с английского А. Киселев

Изготовлено в Росии. Изготовитель: ТОО «Спринт Бук».
Место нахождения и фактический адрес
010000, Казахстан, город Астана, район Алматы, Проспект Ракымжан Кошкарбаев, д. 10/1, н. п. 18.

Дата изготовления: 06.2025. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 23.05.25. Формат 60×90/16. Бумага офсетная. Усл. п. л. 16,000.
Тираж 1000. Заказ 0000.

Дэвид Н. Бланк-Эдельман

НАСТОЯЩИЙ SRE: ИНЖИНИРИНГ НАДЕЖНОСТИ ДЛЯ СПЕЦИАЛИСТОВ И ОРГАНИЗАЦИЙ

Вы хотели бы, чтобы в книгах по обеспечению надежности систем изучение начиналось с азов? Мечтаете, чтобы кто-нибудь подробно разъяснил, как стать SR-инженером, научил мыслить как SR-инженер или подсказал, как создать отдел обеспечения надежности в вашей организации?

Книга «Настоящий SRE» решает все эти и другие задачи: в трех взаимосвязанных разделах приводятся важнейшие основы, необходимые для понимания идеи и культуры SRE, советы отдельным людям, как стать настоящим SRE, и руководство для организаций по созданию и развитию успешной практики SRE.

SPRINT
book