



Software Ошибки и компромиссы

при разработке ПО

Томаш Лелек
Джон Скит



MANNING



Software Mistakes and Tradeoffs

HOW TO MAKE GOOD PROGRAMMING DECISIONS

TOMASZ LELEK
JON SKEET



MANNING
SHELTER ISLAND

Software Ошибки и компромиссы

при разработке ПО

Томаш Лелек
Джон Скит



Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018-02

УДК 004.415

Л43

Лелек Томаш, Скит Джон

Л43 Software: Ошибки и компромиссы при разработке ПО. — СПб.: Питер, 2023. — 464 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2320-9

Создание программных продуктов всегда связано с компромиссами. В попытках сбалансировать скорость, безопасность, затраты, время доставки, функции и многие другие факторы можно обнаружить, что вполне разумное дизайнерское решение на практике оказывается сомнительным. Советы экспертов и яркие примеры, представленные в этой книге, научат вас делать правильный выбор в дизайне и проектировании приложений.

Мы будем рассматривать реальные сценарии, в которых были приняты неверные решения, а затем искать пути, позволяющие исправить подобную ситуацию. Томаш Лелек и Джон Скит делятся опытом, накопленным за десятки лет разработки ПО, в том числе рассказывают о собственных весьма поучительных ошибках. Вы по достоинству оцените конкретные советы и практические методы, а также неустаревающие паттерны, которые изменят ваш подход к проектированию.

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

Права на издание получены по соглашению с Manning Publications.

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. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1617299209 англ.
ISBN 978-5-4461-2320-9 рус.

©2022 by Manning Publications Co. All rights reserved.

© Перевод на русский язык ООО «Прогресс книга», 2023

© Издание на русском языке, оформление ООО «Прогресс книга», 2023

© Серия «Библиотека программиста», 2023

Томаш посвящает эту книгу всему сообществу разработчиков, использующих открытый код. Большинство инструментов и архитектур появляется благодаря вашему энтузиазму и творческому участию. Благодаря вам программные продукты развиваются и соответствуют современным требованиям.

*Джон посвящает написанные им главы всем программистам, которые хоть раз ломали голову над проблемой, возникшей из-за часовых поясов или ромбовидных зависимостей.
(А это делало большинство разработчиков...)*

Краткое содержание

https://t.me/it_boooks

Предисловие	18
Благодарности	20
О книге	22
Об авторах	26
Иллюстрация на обложке	27
От издательства	28
Глава 1. Введение	29
Глава 2. Дублирование кода не всегда плохо: дублирование кода и гибкость	46
Глава 3. Исключения и другие паттерны обработки ошибок в коде	75
Глава 4. Баланс между гибкостью и сложностью	112
Глава 5. Преждевременная оптимизация и оптимизация критического пути: решения, влияющие на производительность кода	137
Глава 6. Простота и затраты на обслуживание API	170
Глава 7. Эффективная работа с датой и временем	198

8 Краткое содержание

Глава 8. Локальность данных и использование памяти	258
Глава 9. Сторонние библиотеки: используемые библиотеки становятся кодом	290
Глава 10. Целостность и атомарность в распределенных системах	323
Глава 11. Семантика доставки в распределенных системах	347
Глава 12. Управление версиями и совместимостью.	374
Глава 13. Современные тенденции разработки и затраты на сопровождение кода.	433

Оглавление

https://t.me/it_boooks

Предисловие	18
Благодарности	20
О книге	22
Для кого эта книга.	22
Структура книги.	22
О коде	24
Форум liveBook	25
Об авторах	26
Иллюстрация на обложке	27
От издательства	28
Словарь паттернов проектирования	28
Глава 1. Введение	29
1.1. Последствия каждого решения и паттерна	30
1.1.1. Решения в модульном тестировании.	31
1.1.2. Соотношение модульных и интеграционных тестов	32
1.2. Программные паттерны проектирования и почему они работают не всегда	34
1.2.1. Измерение скорости выполнения.	39

1.3. Архитектурные паттерны проектирования и почему они работают не всегда	41
1.3.1. Масштабируемость и эластичность	41
1.3.2. Скорость разработки.	42
1.3.3. Сложность микросервисов	43
Итоги.	45
Глава 2. Дублирование кода не всегда плохо: дублирование кода и гибкость	46
2.1. Общий код в кодовых базах и дублирование	47
2.1.1. Добавление нового бизнес-требования, для которого дублирование кода необходимо	49
2.1.2. Реализация нового бизнес-требования	50
2.1.3. Оценка результата	51
2.2. Библиотеки и совместное использование кода в кодовых базах	52
2.2.1. Оценка компромиссов и недостатков совместно используемых библиотек	53
2.2.2. Создание совместно используемой библиотеки.	54
2.3. Выделение кода в отдельный микросервис.	55
2.3.1. Компромиссы и недостатки отдельного сервиса	58
2.3.2. Выводы о выделении отдельных сервисов	62
2.4. Улучшение слабой связанности за счет дублирования кода	63
2.5. Проектирование API с наследованием для сокращения дублирования.	67
2.5.1. Выделение базового обработчика запросов	68
2.5.2. Наследование и сильная связанность	70
2.5.3. Компромиссы между наследованием и композицией	72
2.5.4. Дублирование внутреннее и ситуативное.	73
Итоги.	74
Глава 3. Исключения и другие паттерны обработки ошибок в коде.	75
3.1. Иерархия исключений	77
3.1.1. Универсальный и детализированный подход к обработке ошибок	78

3.2. Лучшие паттерны для обработки исключений в собственном коде .	81
3.2.1. Обработка проверяемых исключений в общедоступном API.	82
3.2.2. Обработка непроверяемых исключений в общедоступном API.	83
3.3. Антипаттерны в обработке исключений	85
3.3.1. Закрытие ресурсов при возникновении ошибки	87
3.3.2. Антипаттерн использования исключений для управления программной логикой	89
3.4. Исключения из сторонних библиотек.	90
3.5. Исключения в многопоточных средах.	93
3.5.1. Исключения в асинхронной программной логике с API обещаний.	96
3.6. Функциональный подход к обработке ошибок с Try.	99
3.6.1. Использование Try в рабочем коде.	103
3.6.2. Объединение Try с кодом, выдающим исключение	105
3.7. Сравнение производительности кода обработки исключений.	106
Итоги.	110
Глава 4. Баланс между гибкостью и сложностью	112
4.1. Мощный, но не расширяемый API.	113
4.1.1. Проектирование нового компонента.	113
4.1.2. Начиная с простого	114
4.2. Возможность предоставления собственной библиотеки метрик	118
4.3. Обеспечение расширяемости API с использованием перехватчиков	121
4.3.1. Защита от непредвиденного использования API перехватчиков	123
4.3.2. Влияние API перехватчиков на производительность	125
4.4. Обеспечение расширяемости API за счет использования прослушивателей	129
4.4.1. Прослушиватели и перехватчики.	130
4.4.2. Неизменяемость архитектуры.	131

4.5. Анализ гибкости API и затраты на обслуживание	133
Итоги	136
Глава 5. Преждевременная оптимизация и оптимизация критического пути: решения, влияющие на производительность кода	137
5.1. Когда преждевременная оптимизация — зло.	138
5.1.1. Создание конвейера обработки учетных записей	139
5.1.2. Оптимизация обработки на основании ложных утверждений	140
5.1.3. Оценка оптимизации производительности	141
5.2. Критические пути в коде	144
5.2.1. Принцип Парето в контексте программных систем	146
5.2.2. Настройка количества параллельных пользователей (поток) для заданного уровня SLA	147
5.3. Словарный сервис с потенциальным критическим путем	148
5.3.1. Получение «слова дня»	149
5.3.2. Проверка существования слова	151
5.3.3. Предоставление доступа к WordsService с использованием сервиса HTTP	151
5.4. Обнаружение критического пути в коде	153
5.4.1. Применение Gatling для создания тестов производительности API.	153
5.4.2. Измерение хронометража кодовых путей с использованием MetricRegistry	157
5.5. Повышение производительности критического пути	159
5.5.1. Создание микротеста JMH для существующего решения	160
5.5.2. Оптимизация проверки с использованием кэширования. . . .	161
5.5.3. Увеличение количества входящих слов в тестах производительности	167
Итоги	169
Глава 6. Простота и затраты на обслуживание API	170
6.1. Базовая библиотека, используемая другими инструментами	171
6.1.1. Создание клиента облачного сервиса	172
6.1.2. Стратегии аутентификации	173

6.1.3. Понимание механизма конфигурации.	175
6.2. Прямое предоставление настроек зависимой библиотеки	179
6.2.1. Конфигурация пакетного инструмента	181
6.3. Абстрагирование настроек зависимой библиотеки.	183
6.3.1. Конфигурация стримингового сервиса	184
6.4. Добавление новой настройки для облачной клиентской библиотеки	186
6.4.1. Добавление новой настройки в пакетный инструмент.	187
6.4.2. Добавление новой настройки в стриминговый сервис	188
6.4.3. Сравнение UX-ориентированности и удобства обслуживания в двух решениях	189
6.5. Удаление настроек в облачной клиентской библиотеке.	190
6.5.1. Удаление настройки из пакетного инструмента.	192
6.5.2. Удаление настроек из стримингового сервиса.	194
6.5.3. Сравнение UX-ориентированности и затрат на обслуживание для двух решений.	196
Итоги.	197
Глава 7. Эффективная работа с датой и временем.	198
7.1. Концепции представления даты и времени	200
7.1.1. Машинное время: моменты времени, эпохи и интервалы.	200
7.1.2. Календарные системы, даты, время и периоды	204
7.1.3. Часовые пояса, UTC и смещения от UTC	211
7.1.4. Концепции даты и времени, вызывающие приступ головной боли	216
7.2. Подготовка к работе с информацией о дате и времени	219
7.2.1. Ограничение объема работ	219
7.2.2. Уточнение требований к дате и времени	221
7.2.3. Использование подходящих библиотек или пакетов	227
7.3. Реализация кода даты и времени	229
7.3.1. Последовательное применение концепций.	229
7.3.2. Отказ от значений по умолчанию в целях улучшения тестируемости	232

7.3.3. Текстовое представление даты и времени	239
7.3.4. Объяснение кода в комментариях	246
7.4. Граничные случаи	249
7.4.1. Арифметические операции с календарями	249
7.4.2. Переходы часовых поясов в полночь.	250
7.4.3. Обработка неоднозначного или пропущенного времени	251
7.4.4. Изменения данных часовых поясов	251
Итоги.	256
Глава 8. Локальность данных и использование памяти	258
8.1. Что такое локальность данных?.	259
8.1.1. Перемещение вычислений к данным	260
8.1.2. Масштабирование обработки с использованием локальности данных.	261
8.2. Секционирование и разбиение данных	263
8.2.1. Автономное секционирование больших данных	263
8.2.2. Секционирование и сегментирование	266
8.2.3. Алгоритмы секционирования	267
8.3. Соединение наборов больших данных из нескольких секций	270
8.3.1. Соединение данных на одной физической машине.	271
8.3.2. Соединение, требующее перемещения данных	273
8.3.3. Оптимизация соединения за счет широковещательной рассылки.	274
8.4. Обработка данных: память и диск	276
8.4.1. Обработка с хранением данных на диске	276
8.4.2. Для чего нужна парадигма MapReduce?	277
8.4.3. Вычисление времени обращения	280
8.4.4. Обработка данных в памяти	281
8.5. Реализация соединений с использованием Apache Spark.	283
8.5.1. Реализация соединения без рассылки	284
8.5.2. Реализация соединения с рассылкой	287
Итоги.	288

Глава 9. Сторонние библиотеки: используемые библиотеки становятся кодом	290
9.1. Импорт библиотеки и ответственность за ее настройки: берегитесь значений по умолчанию	291
9.2. Модели параллельного выполнения и масштабируемость	296
9.2.1. Использование асинхронных и синхронных API	298
9.2.2. Распределенная масштабируемость	301
9.3. Тестируемость.	303
9.3.1. Тестовая библиотека.	304
9.3.2. Тестирование с использованием объектов fake (тестовых двойников) и mock.	306
9.3.3. Набор инструментов интеграционного тестирования	311
9.4. Зависимости сторонних библиотек	312
9.4.1. Предотвращение конфликтов версий	313
9.4.2. Слишком много зависимостей.	315
9.5. Выбор и обслуживание сторонних зависимостей.	316
9.5.1. Первые впечатления	316
9.5.2. Разные подходы к повторному использованию кода.	317
9.5.3. Привязка к производителю	318
9.5.4. Лицензирование	319
9.5.5. Библиотеки и фреймворки	319
9.5.6. Безопасность и обновления	319
9.5.7. Список решений	320
Итоги.	321
Глава 10. Целостность и атомарность в распределенных системах	323
10.1 Источники данных с доставкой «не менее одного раза»	324
10.1.1. Трафик между сервисами с одним узлом	324
10.1.2. Повтор попытки вызова	326
10.1.3. Производство данных и идемпотентность	327
10.1.4. Паттерн CQRS.	330
10.2. Наивная реализация дедупликации	332

10.3. Типичные ошибки при реализации дедупликации в распределенных системах	335
10.3.1. Одноузловый контекст	336
10.3.2. Многоузловый контекст.	338
10.4. Обеспечение атомарности логики для предотвращения ситуации гонки.	341
Итоги.	345
Глава 11. Семантика доставки в распределенных системах.	347
11.1. Архитектура событийно-управляемых приложений	348
11.2. Производители и потребители на базе Apache Kafka.	352
11.2.1. Kafka на стороне потребителя	353
11.2.2. Конфигурация брокеров Kafka	355
11.3. Логика производителя.	356
11.3.1. Выбор между целостностью данных и доступностью для производителя	359
11.4. Код потребителя и разные семантики доставки	362
11.4.1. Ручная фиксация у потребителя	364
11.4.2. Перезапуск от самых ранних или поздних смещений	366
11.4.3. Семантика «фактически ровно один»	369
11.5. Использование гарантий доставки для обеспечения отказоустойчивости.	371
Итоги.	373
Глава 12. Управление версиями и совместимостью.	374
12.1. Версионирование на абстрактном уровне.	375
12.1.1. Свойства версий.	375
12.1.2. Обратная и прямая совместимость	377
12.1.3. Семантическое версионирование	378
12.1.4. Маркетинговые версии	381
12.2. Версионирование для библиотек	381
12.2.1. Совместимость уровня исходного кода, двоичная и семантическая	382
12.2.2. Графы зависимостей и ромбовидные зависимости	391

12.2.3. Снижение последствий от критических изменений	397
12.2.4. Управление библиотеками только для внутреннего пользования.	402
12.3. Версионирование для сетевых API	403
12.3.1. Контекст вызовов сетевых API	404
12.3.2. Ясность для клиента	405
12.3.3. Популярные стратегии версионирования	407
12.3.4. Дополнительные аспекты версионирования	413
12.4. Версионирование для хранилищ данных	417
12.4.1. Краткое введение в Protocol Buffers	418
12.4.2. Что является критическим изменением?	420
12.4.3. Миграция данных в системе хранения.	421
12.4.4. В ожидании неожиданного	425
12.4.5. Разделение API и представлений хранения данных	427
12.4.6. Оценка форматов хранения.	430
Итоги.	431

Глава 13. Современные тенденции разработки и затраты

на сопровождение кода.	433
13.1. Когда использовать фреймворки внедрения зависимостей.	434
13.1.1. Самостоятельная реализация внедрения зависимостей	435
13.1.2. Использование фреймворка внедрения зависимостей.	438
13.2. Когда применяется реактивное программирование	441
13.2.1. Создание однопоточной обработки с блокированием	442
13.2.2. Использование CompletableFuture	444
13.2.3. Реализация реактивного решения.	447
13.3. Когда применяется функциональное программирование	449
13.3.1. Создание функционального кода на нефункциональном языке	450
13.3.2. Хвостовая рекурсия	453
13.3.3. Использование неизменяемости	454
13.4. Отложенное и немедленное вычисление	456
Итоги.	459

Предисловие

Работа всех участников процесса создания программных продуктов полна компромиссов. Нам часто приходится действовать в условиях ограничений времени, бюджета и информации. А это значит, что решения о продукте, принятые сегодня, будут иметь последствия завтра: затраты на обслуживание, негибкость продукта при необходимости внесения изменений, ограниченная производительность при масштабировании и многие другие. Важно, что каждое решение принимается в определенном контексте. Легко оценивать прошлые шаги без полного знания контекста, в котором они были сделаны. При этом чем больше информации и глубже анализ, выполняемый в момент принятия решений, тем лучше понимание компромиссов, заложенных в эти решения.

За свою карьеру мы имеем дело со многими программными продуктами и изучаем компромиссы, которые в них заложены. В процессе работы Томаш начал вести журнал с описанием обстоятельств, в которых принималось то или иное решение. Каким был контекст? Какие существовали альтернативы? Как оценить конкретное решение? Наконец, каким оказался результат? Предусмотрели ли мы все возможные компромиссы конкретного решения? Столкнулись ли с неожиданностями? Как выяснилось, этот персональный список усвоенных уроков в действительности представлял собой те проблемы и решения, с которыми сталкиваются многие разработчики. Тогда Томаш решил, что этими знаниями стоит поделиться с миром. Так родилась идея этой книги.

Мы хотели рассказать о том, что мы усвоили на личном опыте работы с разными программными системами: монолитами, микросервисами, обработкой больших данных, библиотеками и многими другими. В книге тщательно анализируются решения, компромиссы и ошибки реальных систем. Представляя эти паттерны, ошибки и уроки, мы надеемся расширить ваш контекст и вооружить вас более

эффективными инструментами, которые помогут принимать лучшие решения в повседневной работе. Если вы будете сразу видеть потенциальные проблемы и ограничения архитектуры, то сэкономите массу времени и денег в будущем. Мы не пытаемся дать однозначные ответы. Сложные задачи часто можно решать разными способами. Мы опишем некоторые из таких задач и будем задавать вопросы без однозначных ответов. У всех решений есть свои достоинства и недостатки, и мы проанализируем их. Каждое решение приводит к своим компромиссам, и вы сами должны решить, какие из них лучше всего подходят для вашего контекста.

Благодарности

Работа над книгой требует больших усилий. Тем не менее благодаря издательству Manning она стала настоящим удовольствием.

Прежде всего я благодарю свою жену Малгожату. Ты всегда поддерживала меня, прислушивалась к моим идеям и проблемам. Ты помогла мне сосредоточиться на книге.

Я благодарю своего редактора в Manning Дуга Раддера (Doug Rudder). Спасибо за совместную работу — твои комментарии и обратная связь были бесценны. С твоей помощью я прокачал свои писательские навыки до нового уровня. Спасибо всем остальным сотрудникам Manning, которые участвовали в создании и продвижении книги. Это действительно была командная работа. Отдельное огромное спасибо выпускающему редактору Дейдре Хайем (Deirdre Hiam); литературному редактору Кристиану Берку (Christian Berk); ревьюеру Михаэле Батинич (Mihaela Batinic) и корректору Джейсону Эверетту (Jason Everett).

Также хотелось бы поблагодарить рецензентов, которые не пожалели времени, чтобы прочитать мою рукопись на разных стадиях ее создания, и предоставили бесценную обратную связь — ваши предложения помогли мне улучшить книгу: Алекс Сез (Alex Saez), Александр Вейер (Alexander Weiher), Андреас Сакко (Andres Sacco), Эндрю Эленески (Andrew Eleneski), Энди Кирш (Andy Kirsch), Конор Редмонд (Conor Redmond), Косимо Атанаси (Cosimo Atanasi), Дэйв Корун (Dave Corun), Джордж Томас (George Thomas), Жиль Ячелини (Gilles Iachellini), Грегори Варгезе (Gregory Varghese), Хьюго Крус (Hugo Cruz), Иоханнес Вервийнен (Johannes Verwijnen), Джон Гатри (John Guthrie), Джон Генри Галино (John Henry Galino), Джонни Слос (Johnny Slos), Максим Прохоренко (Maksym Prokhorenko), Марк-Оливер Шееле (Marc-Oliver Scheele), Нельсон

Гонсалес (Nelson González), Оливер Кортен (Oliver Korten), Паоло Брунасти (Paolo Brunasti), Рафаэль Авила Мартинес (Rafael Avila Martinez), Раджиш Моханан (Rajesh Mohanan), Роберт Траусмут (Robert Trausmuth), Роберто Касадеи (Roberto Casadei), Со Фай Фон (Sau Fai Fong), Шон Лам (Shawn Lam), Спенсер Маркс (Spencer Marks), Василь Борис (Vasile Boris), Винсент Делкойн (Vincent Delcoigne), Витош Дойнов (Vitosh Doynov), Уолтер Стоунбернер (Walter Stoneburner) и Уилл Прайс (Will Price).

Отдельное спасибо редактору-консультанту Джин Боярски (Jeanne Boyarsky) за подробный разбор текста с технической точки зрения.

Эта книга стала результатом всех профессиональных решений, которые принимал я и те, с кем я общался в ходе работы. Я встречал многих людей, которые сформировали меня как программиста и положительно повлияли на мою карьеру. Мне повезло познакомиться и работать с ними в начале карьеры. Хочу поблагодарить всех своих коллег из Schibsted, Allegro, DataStax и Dremio. Некоторые из них заслуживают особой благодарности:

- *Павел Волошин (Paweł Wołoszyn)* — за то, что он превосходный университетский лектор, научивший меня тому, что программирование способно кардинально изменить мир.
- *Анджей Гжесик (Andrzej Grzesik)* — за то, что вдохновил меня на стремление к достижению амбициозных целей.
- *Матеуш Квасьневский (Mateusz Kwaśniewski)* — за то, что разжег во мне искру жажды к знаниям.
- *Лукаш Банцеровский (Łukasz Banceroński)* — за то, что указал мне направление и сформировал мою карьеру в сфере JVM.
- *Ярослав Палка (Jarosław Palka)* — за то, что поверил в меня и обеспечил пространство для экспериментов и обучения.
- *Александр Дутра (Alexandre Dutra)* — за то, что подавал пример и демонстрировал высочайшую рабочую этику.

— *Томаш Лелек*

Спасибо всем, кому я годами досаждал рассказами о часовых поясах, особенно моей многострадальной семье. Мои коллеги в Google, а также участники проекта с открытым кодом Noda Time и других очень помогли продумать аспекты, которые я изложил в этой книге.

— *Джон Скит*

О книге

Книга «Software: Ошибки и компромиссы при разработке ПО» была задумана как описание проблем, встречающихся в реальных системах. Мы постарались проанализировать каждую ситуацию в разных контекстах и рассмотреть все возможные компромиссы. Кроме того, в книге представлены некоторые неочевидные ошибки, способные значительно повлиять на системы в разных аспектах (не только с точки зрения правильности).

ДЛЯ КОГО ЭТА КНИГА

Книга предназначена для разработчиков, которые хотят изучить паттерны, используемые в реальных системах, и связанные с ними компромиссы. Также она учит избегать неочевидных ошибок. Книга начинается с низкоуровневых решений, что очень полезно для начинающих разработчиков. Затем рассматриваются более сложные темы — это пригодится даже опытному специалисту. Большинство примеров, паттернов и фрагментов кода написаны на Java, но сами решения не привязаны к этому языку.

СТРУКТУРА КНИГИ

Книга состоит из 13 глав. В первой главе приводится общий обзор использованных в книге методов анализа компромиссов. Остальные главы относительно независимы друг от друга, в них рассматриваются разные вопросы разработки. Чтобы извлечь наибольшую пользу из книги, мы рекомендуем читать ее по

порядку. Тем не менее, если вас интересует конкретный вопрос, переходите к нужной главе.

- В главе 1 представлен подход, используемый в книге для анализа компромиссов в определенном контексте. В ней приводятся примеры компромиссов на уровнях программной архитектуры, кода и контроля качества.
- Глава 2 показывает, что дублирование кода не всегда является антипаттерном. В ней рассматриваются различные архитектуры и анализируется их влияние на слабую связанность систем. Глава завершается вычислением затрат на координацию внутри команд и между ними по закону Амдала.
- В главе 3 описаны стратегии обработки аномальных ситуаций в коде. В ней разобраны сценарии использования как проверяемых, так и непроверяемых исключений. Вы научитесь разрабатывать стратегии обработки исключений для общедоступных API (библиотек). Наконец, в ней рассматриваются компромиссы между подходами обработки исключений, применяемыми в объектном и функциональном программировании.
- Глава 4 учит выдерживать баланс между сложностью кода и API. Она показывает, что эволюция кода в одном из направлений часто влияет на другие направления.
- Глава 5 объясняет, что преждевременная оптимизация не всегда плоха. С правильным инструментарием и определенными условиями SLA можно найти критический путь и оптимизировать его. Кроме того, она демонстрирует, что принцип Парето полезен, чтобы сконцентрировать усилия по оптимизации в нужной части системы.
- В главе 6 показано, как проектировать API с качественным UX. Она показывает, что удобство UX характеризует не только пользовательские, но и программные интерфейсы: REST API, средства командной строки и т. д. Однако глава также показывает, что за удобство UX приходится расплачиваться повышением затрат на обслуживание.
- В главе 7 рассматриваются болезненные вопросы обработки информации даты и времени. Если учесть, как часто данные содержат по крайней мере некоторые элементы даты и времени — например, дату рождения или временную метку записи в журнале, возможностей для возникновения ошибок более чем достаточно. С этими проблемами можно справиться, но они требуют особого внимания.
- Глава 8 объясняет, почему локальность данных играет важную роль в обработке больших данных. Она демонстрирует необходимость алгоритмов разбиения, обеспечивающих распределение данных и трафика.
- Глава 9 показывает, что используемые вами библиотеки становятся вашим кодом. В ней рассмотрены проблемы и компромиссы, которые необходимо учитывать при импортировании сторонней библиотеки в кодовую базу. Нако-

нец, глава пытается ответить на вопрос, стоит ли импортировать библиотеку или лучше заново реализовать ее отдельные части.

- Глава 10 посвящена компромиссу между целостностью данных и атомарностью в распределенных системах. В ней анализируются возможные ситуации гонки в таких системах и влияние идемпотентности на способы проектирования систем.
- Глава 11 объясняет семантику доставки в распределенных системах. Она помогает понять семантику «не менее одного», «не более одного» и «фактически ровно один».
- В главе 12 вы узнаете, как программные системы, API и хранимые данные эволюционируют со временем и как при этом сохранять совместимость с другими системами.
- Глава 13 демонстрирует, что не всегда разумно гнаться за новейшими тенденциями в IT-отрасли. В ней анализируются некоторые популярные паттерны и фреймворки (такие, как реактивное программирование), а также обсуждается их применение в конкретных контекстах.

О КОДЕ

Книга содержит множество примеров исходного кода как в нумерованных листингах, так и в тексте. В обоих случаях исходный код форматируется **моноширинным шрифтом**, в отличие от обычного текста. Иногда для кода также применяется **жирный шрифт**, чтобы выделить фрагменты, изменившиеся по сравнению с предыдущими шагами, — например, при добавлении новой функциональности в существующую строку кода.

Во многих случаях оригинальная версия исходного кода переформатируется; добавляются разрывы строк и измененные отступы, чтобы код помещался на странице. Иногда даже этого оказывается недостаточно и в листинги включаются маркеры продолжения строк (**↪**). Также из исходного кода часто удаляются комментарии, если код описывается в тексте.

Исходный код в книге форматирован автоматизированным плагином в соответствии с рекомендациями Google по оформлению кода. Многие листинги снабжены примечаниями, выделяющими важные концепции. Каждая глава представлена отдельной папкой в репозитории. Для всего кода, использованного в книге, написаны многочисленные модульные и интеграционные тесты, обеспечивающие его качество. Не все тесты приводятся в листингах книги. Вы можете как запустить тесты, так и прочитать их код по отдельности, чтобы лучше понять конкретную часть логики. Все инструкции по импортированию и запуску примеров содержатся в файле **README.md** в репозитории.

Исполняемые фрагменты кода можно загрузить из версии liveBook (электронной) по адресу <https://livebook.manning.com/book/software-mistakes-and-tradeoffs>. Полный код примеров книги доступен для загрузки на сайте Manning по адресам <https://www.manning.com/books/software-mistakes-and-tradeoffs> и https://github.com/tomekl007/manning_software_mistakes_and_tradeoffs.

ФОРУМ LIVEBOOK

Приобретая книгу, вы получаете бесплатный доступ к liveBook, платформе Manning для чтения в интернете. С помощью эксклюзивных средств liveBook можно добавлять комментарии к книгам — глобально или к отдельным разделам/абзацам. Кроме того, можно делать заметки для себя, задавать технические вопросы и отвечать на них, получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, перейдите по ссылке <https://livebook.manning.com/book/software-mistakes-and-tradeoffs/discussion>. Узнать больше о форумах Manning и правилах поведения на них можно на странице <https://livebook.manning.com/discussion>.

Manning соблюдает обязательство перед читателями по предоставлению площадки для содержательной дискуссии между читателями и между читателями и автором. Со стороны автора отсутствуют обязательства о конкретной доле участия, и его вклад в работу форума остается добровольным (и неоплачиваемым). Задавайте автору сложные вопросы, чтобы его интерес не потерялся! Форум и архивы предыдущих обсуждений будут доступны на веб-сайте издателя, пока книга продолжает издаваться.

Об авторах

Томаш Лелек

За свою карьеру разработчика Томаш имел дело с разными сервисами, архитектурами и языками программирования (прежде всего для JVM). У него есть реальный опыт работы с монолитными и микросервисными архитектурами. Томаш проектировал системы, обрабатывающие запросы десятков миллионов уникальных пользователей и сотни тысяч операций в секунду. В частности, он работал над следующими проектами:

- Микросервисная архитектура с CQRS (на базе Apache Kafka).
- Автоматизация маркетинга и обработка потоков событий.
- Обработка больших данных в Apache Spark и Scala.

Сейчас Томаш работает в Dremio, где помогает создавать современные решения для хранения данных. До этого он работал в DataStax, где строил различные продукты для Cassandra Database. Он создавал инструменты для тысяч разработчиков, которые больше всего ценят проектирование API, производительность и удобство UX. Он внес свой вклад в разработку Java-Driver, Cassandra Quarkus, соединителя Cassandra-Kafka и Stargate.

Джон Скит

Джон — специалист по выстраиванию отношений с разработчиками в Google, в настоящее время работающий над библиотеками Google Cloud Client Libraries для .NET. Его вклад в сообщество с открытым кодом включает библиотеку даты и времени Noda Time для .NET (<https://nodatime.org>). Вероятнее всего, он известен благодаря своим публикациям на сайте Stack Overflow. Джон — автор книги «C# in Depth» (издательство Manning). Также он участвовал в работе над книгами «Groovy in Action» и «Real-World Functional Programming». Джон интересуется API даты/времени и версионированием — многим эти увлечения кажутся в лучшем случае необычными.

Иллюстрация на обложке

Иллюстрация под названием Groenlandaise («Гренландка»), помещенная на обложку, взята из вышедшего в 1797 году каталога национальных костюмов, составленного Жаком Грассе де Сен-Совером. Каждая иллюстрация этого каталога тщательно прорисована и раскрашена от руки. В прежние времена по одежде человека можно было легко определить, где он живет и какова его профессия или положение. Manning отдает дань изобретательности и инициативности компьютерных технологий, используя для своих изданий обложки, демонстрирующие богатое вековое разнообразие региональных культур, оживающее на изображениях из собраний, подобных этому.

От издательства

На момент сдачи книги в печать все приведенные в ней URL-ссылки работают, однако доступ к некоторым сайтам ограничен на территории РФ.

В книге используются паттерны проектирования. При первом упоминании паттерна в скобках дается его английское название, чтобы читатели смогли без труда отследить его в листингах. Для удобства ниже приведен Словарь паттернов проектирования, встречающихся в данной книге.

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

Мы будем рады узнать ваше мнение!

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

СЛОВАРЬ ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ

Декоратор	Decorator
Наблюдатель	Observer
Одиночка	Singleton
Заместитель	Proxy
Прерыватель	Circuit Breaker
Прототип	Prototype
Стратегия	Strategy
Строитель	Builder
Фабрика	Factory

1

Введение

https://t.me/it_boooks

В этой главе:

- ✓ Важные компромиссы в эксплуатируемых системах.
- ✓ Последствия модульного и интеграционного тестирования.
- ✓ Паттерны проектирования и программирования не решают всех проблем.

При проектировании кода, API и системных архитектур приходится принимать решения, влияющие на обслуживание, производительность, расширяемость и многие другие факторы. Почти всегда решение пойти в одном направлении ограничивает возможность развиваться в другом. Чем дольше живут системы, тем сложнее изменить их архитектуру и отказаться от предыдущих решений. Компромиссы в проектировании и программировании, представленные в этой книге, связаны с выбором из двух и более направлений, в которых может развиваться система. Важно понимать, что, какой бы вариант вы ни выбрали, придется иметь дело со всеми его плюсами и минусами.

Команда должна принимать эти непростые решения в зависимости от контекста, времени выхода на рынок, соглашения об уровне обслуживания (SLA, Service-Level Agreement) и других факторов. Мы опишем некоторые компромиссы, на которые приходится идти в рабочих системах, и сравним их с альтернативными подходами. Надеемся, что после прочтения этой книги вы научитесь внимательно оценивать свои ежедневные решения в проектировании. Это позволит вам принимать их осознанно, учитывая все «за» и «против».

Первая часть книги посвящена низкоуровневым проектным решениям, которые вынужден принимать каждый программист при работе с кодом и API. Вторая часть освещает системы более широко — в ней рассматривается архитектура и потоки данных между компонентами. Также мы поговорим о компромиссах, на которые приходится идти при работе с распределенными системами.

В следующих разделах продемонстрирован подход к анализу компромиссов, принятый в нашей книге. Сначала мы сосредоточимся на компромиссах, с которыми сталкивается любой разработчик: балансе между модульными, интеграционными, сквозными и другими видами тестов. В реальных условиях программный продукт приносит пользу в течение ограниченного периода. Поэтому нужно решить, на какие тесты выделить больше времени — модульные, интеграционные, сквозные или другие. Мы проанализируем плюсы и минусы увеличения количества тестов конкретного типа.

Затем мы рассмотрим зарекомендовавший себя паттерн Одиночка (Singleton) и объясним, как его полезность меняется в зависимости от контекста (однопоточного или многопоточного). Наконец, рассмотрим архитектурные компромиссы более высокого уровня: выбор между микросервисной и монолитной архитектурой.

Следует заметить, что зачастую архитектуру невозможно описать как чисто монолитную или чисто микросервисную. На практике распространен гибридный подход: часть функционала реализована в виде сервисов, тогда как другие части системы могут существовать в монолитном виде. Например, когда унаследованная система монолитна и лишь малая ее часть перемещена в микросервисную архитектуру. Также в проекте, создаваемом с нуля, вполне разумно начать с одного приложения и не разбивать его на микросервисы, если это сопряжено с ощутимыми затратами. Мы кратко проанализируем компромиссы между микросервисами и монолитами. Рекомендуем частично применять эту аргументацию в актуальном контексте, даже если это гибридная архитектура.

В следующих разделах представлен подход, который используется во всех главах: решение задачи в конкретной ситуации, затем анализ альтернативного варианта и, наконец, добавление контекста, подразумевающего компромиссы и окончательные решения. Плюсы и минусы каждого из них проанализированы в конкретном контексте. В следующих главах компромиссные решения будут рассмотрены более подробно.

1.1. ПОСЛЕДСТВИЯ КАЖДОГО РЕШЕНИЯ И ПАТТЕРНА

Цель книги — продемонстрировать различные компромиссы и ошибки в проектировании и программировании. При разборе проектных решений и компромиссов

я буду исходить из того, что вы пишете достаточно хороший код. Когда качество кода становится приемлемым, необходимо определить направление, в котором он должен развиваться.

Чтобы понять логику каждой главы, начнем с изучения компромиссов между двумя самыми полезными и очевидными приемами тестирования, которые должны применяться в коде: интеграционными и модульными тестами. Главная цель — обеспечить покрытие практически каждого пути тестами этих двух видов. Зачастую на практике это нецелесообразно, потому что время, выделенное для написания и тестирования кода, ограничено. Следовательно, выбор соотношения модульного и интеграционного тестирования — это стандартный компромисс, на который придется пойти.

1.1.1. Решения в модульном тестировании

При написании тестов необходимо решить, какую часть кода тестировать. Возьмем простой компонент, для которого нужно организовать модульное тестирование. Допустим, компонент `SystemComponent` предоставляет один открытый метод API: `publicApiMethod()`. Другие методы скрываются от клиентов приватным модификатором доступа. В следующем листинге приведен код этого сценария.

Листинг 1.1. Компонент для модульного тестирования

```
public class SystemComponent {

    public int publicApiMethod() {
        return privateApiMethod();
    }

    private int privateApiMethod() {
        return complexCalculations();
    }

    private int complexCalculations() {
        // Сложная логика
        return 0;
    }
}
```

Здесь мы должны решить, стоит ли включать в модульное тестирование `complexCalculations()` или же оставить приватный метод скрытым. Такой модульный тест работает по принципу «черного ящика» и покрывает только открытый API. Обычно этого достаточно. Но иногда приватные методы содержат сложную логику, которую также стоит протестировать. В таких ситуациях можно понизить модификатор доступа `complexCalculations()`. Этот подход продемонстрирован в следующем листинге.

Листинг 1.2. Открытый уровень видимости компонента для модульного тестирования

```
@VisibleForTesting
public int complexCalculations() {
    // Сложная логика
    return 0;
}
```

Переходя на открытый уровень видимости, вы разрешаете себе написать модульный тест для части API, которая не должна быть открытой. Открытый метод будет видимым для клиентов API, и появится риск того, что они будут напрямую использовать этот API. В листинге аннотация `@VisibleForTesting` (см. <http://mng.bz/y4wq>) служит только для информационных целей. Ничто не мешает пользователям вызвать открытый метод API. Если они не заметят эту аннотацию, то могут проигнорировать ее.

Вы можете использовать любой из двух подходов к модульному тестированию, рассмотренных в этом разделе. Последний обеспечивает большую гибкость, но затраты на обслуживание могут возрасти. Возможно, вы выберете промежуточное решение. Для этого можно сделать код пакетно-приватным. Иначе говоря, когда тесты находятся в одном пакете с рабочим кодом, делать его открытым не нужно, но можно использовать эти методы в тестовом коде.

1.1.2. Соотношение модульных и интеграционных тестов

При тестировании логики нужно определиться с соотношением интеграционных и модульных тестов в системе. Часто выбор одного направления снижает возможность развиваться в другом. Кроме того, такое ограничение может зависеть от времени начала разработки системы.

Так как время разработки функционала обычно ограничено, нужно выбрать, чему посвятить большую его часть — модульным или интеграционным тестам. Тестирование реальных систем должно включать обе эти разновидности, и мы должны определить их соотношение.

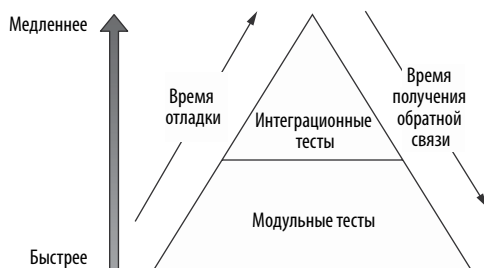


Рис. 1.1. Интеграционные/модульные тесты и время (скорость), необходимое для их выполнения

У каждого вида тестов есть достоинства и недостатки, и выбор между ними становится типичным компромиссом при написании кода. Модульные тесты работают быстрее и обеспечивают более быстрый отклик, так что процесс отладки часто ускоряется. На рис. 1.1 представлены достоинства и недостатки обоих тестов.

Диаграмма на рис. 1.1 имеет форму пирамиды, потому что обычно в программных системах больше модульных тестов, чем интеграционных. Модульные тесты обеспечивают практически мгновенную обратную связь для разработчика, что повышает производительность. Кроме того, они быстрее выполняются и сокращают время отладки кода. Если обеспечить полное покрытие кодовой базы модульными тестами, то при появлении новой ошибки проблема с большой вероятностью будет перехвачена одним из них. Ее можно будет обнаружить на уровне метода, который покрывается конкретным модульным тестом.

С другой стороны, если в системе отсутствуют интеграционные тесты, исчезает возможность анализа связей между компонентами и их объединением в систему. У вас будут хорошо протестированные алгоритмы, но без проверки общей картины. В результате может получиться система, которая правильно функционирует на низком уровне, но без тестирования компонентов невозможно оценить ее работу на более высоком уровне. В реальной жизни код должен сочетать модульные и интеграционные тесты.

Важно заметить, что рис. 1.1 учитывает только один аспект тестирования: время выполнения и, следовательно, время получения обратной связи. В реально эксплуатируемых системах существуют другие уровни тестирования. В них могут быть сквозные тесты, которые осуществляют комплексную проверку бизнес-сценариев. Возможно, в более сложных архитектурах для обеспечения этой бизнес-функциональности придется запустить N сервисов, взаимодействующих друг с другом. У таких тестов обычно медленное время отклика из-за лишних затрат ресурсов на подготовку тестовой инфраструктуры. С другой стороны, они дают более высокую уверенность относительно сквозной логики и правильности системы. Сравнение таких тестов с модульными и интеграционными тестами можно производить по разным параметрам — например, насколько хорошо они проверяют систему в целом, как показано на рис. 1.2.

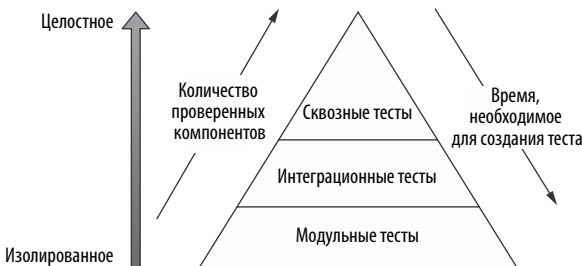


Рис. 1.2. Интеграционные/модульные тесты и время (скорость), необходимое для их выполнения

Так как модульные тесты выполняются изолированно, они не дают сколько-нибудь значимой информации о других компонентах системы и их взаимодействии между собой. Интеграционные тесты пытаются проверять другие компоненты и взаимодействия между ними. Но, как правило, они не охватывают многочисленные (микро-) сервисы, обеспечивающие заданную бизнес-функциональность. Наконец, хотя сквозные тесты проверяют целостность работы системы, количество проверяемых компонентов может быть значительным, так как придется запускать всю инфраструктуру, которая может включать N микросервисов, баз данных, очередей и т. д.

Другое измерение (ресурс), которое необходимо принять во внимание, — время на создание тестов. За короткое время можно создать много модульных тестов, так как они относительно просты в разработке. На проектирование интеграционных тестов часто уходит больше времени. Наконец, сквозные тесты требуют значительных начальных вложений для построения необходимой для них инфраструктуры.

На практике ресурсы ограничены (например, бюджет и время), и приходится максимизировать качество программного продукта с учетом этих условий. Однако покрытие кода тестами позволяет создать более качественный продукт и сократить число багов в релизе. Кроме того, это упрощает дальнейшее обслуживание продукта. Для этого нужно выбрать типы тестов и их количество, а также найти соотношение между модульными, интеграционными и сквозными тестами из-за ограниченности ресурсов. Анализируя разные параметры, сильные и слабые стороны конкретной разновидности тестов, можно принимать более рациональные решения.

Важно, что тестирование увеличивает время разработки. Чем больше тестов мы хотим провести, тем больше времени мы потратим. Иногда трудно реализовать хорошие сквозные тесты, если не планировать их в пределах заданного срока. Таким образом, некоторые виды тестов необходимо планировать так же, как добавление нового функционала в программу, — заранее, а не задним числом.

1.2. ПРОГРАММНЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ И ПОЧЕМУ ОНИ РАБОТАЮТ НЕ ВСЕГДА

Паттерны проектирования — Строитель (Builder), Декоратор (Decorator), Прототип (Prototype) и многие другие — появились много лет назад. Они представляют проверенные на опыте решения многих известных задач. Я настоятельно рекомендую изучить эти паттерны (см. книгу «Design Patterns: Elements of Reusable Object-Oriented Software»¹) и использовать их в коде, чтобы сделать

¹ Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. «Паттерны объектно-ориентированного проектирования». СПб, издательство «Питер».

его более простым в обслуживании и масштабировании — да и просто более качественным. С другой стороны, их следует применять с осторожностью, потому что реализация этих паттернов сильно зависит от контекста. Как вы уже поняли, я пытаюсь показать, что каждое решение в программном продукте подразумевает компромиссы и имеет последствия.

Чтобы понять компромиссы на уровне программного кода, я продемонстрирую паттерн Одиночка (<https://refactoring.guru/design-patterns/singleton>). Он был введен как механизм совместного использования состояния всеми компонентами. Одиночка — единственный экземпляр, существующий на протяжении всего срока жизни приложения. К нему обращаются все остальные классы. Допустим, вам нужно создать приватный конструктор, чтобы предотвратить создание нового экземпляра. Реализовать паттерн Одиночка для этого несложно, как показывает следующий листинг.

Листинг 1.3. Реализация паттерна Одиночка

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Есть только один способ получить экземпляр Одиночки: через метод `getInstance()`, возвращающий единственный экземпляр, который может безопасно использоваться разными компонентами. Предполагается, что каждый раз, когда коду на стороне вызова нужно обратиться к одиночному экземпляру, он делает это через `getInstance()`. Далее мы рассмотрим другой сценарий использования, который не требует применения этого метода каждый раз при обращении к экземпляру. На первый взгляд паттерн позволяет быстро добиться успеха; вы получаете возможность совместно использовать код через глобальный одиночный экземпляр. Казалось бы, в чем компромисс?

Рассмотрим этот паттерн в другом контексте. Что произойдет, если применить его в многопоточной среде? Если сразу несколько потоков вызовут `getInstance()` одновременно, может возникнуть состояние гонки (race condition). В этой ситуации код создаст два экземпляра Одиночки. Разумеется, это нарушит инварианты паттерна и может привести к сбою системы. Чтобы этого избежать, необходимо добавить синхронизацию перед выполнением логики инициализации, как показано в следующем листинге.

Листинг 1.4. Синхронизация для потоково-безопасного паттерна Одиночка

```

public class SystemComponentSingletonSynchronized {
    private static SystemComponent instance;

    private SystemComponentSingletonSynchronized() {}

    public static synchronized SystemComponent getInstance() { ← Начинает блок
        if (instance == null) {                                     синхронизации
            instance = new SystemComponent();
        }

        return instance;
    }
}

```

Блок `synchronized` предотвращает обращение к этой логике из двух потоков. Все потоки, кроме одного, блокируются и ожидают логики инициализации. На первый взгляд все работает, как ожидалось. Но Одиночка в многопоточном сценарии может значительно снизить производительность кода, что важно учитывать, если она первоочередная.

Инициализация — первый процесс, в ходе которого несколько потоков должны блокироваться и ожидать. А после создания одиночного экземпляра все обращения к нему должны синхронизироваться. Одиночка может вызвать конкуренцию потоков (<http://mng.bz/M2nn>), что создаст серьезные риски для производительности. Это происходит, когда имеется общий экземпляр объекта, к которому одновременно обращаются несколько потоков.

Синхронизированный метод `getInstance()` позволяет только одному потоку войти в критическую секцию кода, тогда как остальные вынуждены ожидать снятия блокировки. Когда один поток выходит из критической секции, следующий в очереди может обратиться к ней. Недостаток такого подхода в том, что он создает необходимость синхронизации и может значительно замедлить программу. В двух словах, каждый раз, когда в коде выполняется синхронизируемый вызов, это может привести к дополнительным затратам.

Из примера можно сделать вывод, что существует компромисс между производительностью кода при использовании Одиночки в однопоточном или многопоточном контексте. Но для нас важен контекст, в котором код выполняется. Если он работает без параллельного выполнения или одиночный экземпляр не используется совместно разными потоками, компромисс не проявляется. Но при совместном использовании Одиночки необходимо обеспечить его потоковую безопасность, что потенциально влияет на производительность. Знание этого компромисса позволит принимать рациональные решения, касающиеся архитектуры и кода.

Если окажется, что у выбранного варианта больше минусов, чем плюсов, решение можно изменить. Так, в примере с Одиночкой ситуацию можно улучшить, применив один из двух паттернов.

В первом варианте используется метод блокировки с двойной проверкой. Этот способ отличается тем, что перед входом в критическую (синхронизированную) секцию необходимо проверить экземпляр на наличие `null`. Если условие выполняется, можно продолжать вход в критическую секцию, в противном случае в этом нет необходимости — нужно просто вернуть существующий одиночный экземпляр. Эта реализация блокировки представлена в следующем листинге.

Листинг 1.5. Блокировка с двойной проверкой для паттерна Одиночка

```
private volatile static SystemComponent instance;

public static SystemComponent getInstance() {
    if (instance == null) {
        synchronized (ThreadSafeSingleton.class) {
            if (instance == null) {
                instance = new SystemComponent();
            }
        }
    }
    return instance;
}
```

← Если экземпляр не содержит `null`, не входить в критическую секцию

Применение этого паттерна значительно снижает необходимость в синхронизации и уровень конкуренции между потоками. Эффект синхронизации будет наблюдаться только при запуске, когда каждый поток пытается инициализировать одиночный экземпляр.

Второй паттерн, который можно применить, — привязка к потоку. Он позволяет закрепить состояние за конкретным потоком. Однако при этом необходимо понимать, что паттерн Одиночка перестает существовать на уровне глобального приложения — на каждый поток будет приходиться один экземпляр объекта. Если в приложении существуют N потоков, то в нем будут использоваться N экземпляров.

При использовании данного паттерна каждый поток в коде владеет экземпляром объекта, видимым для конкретного потока и привязанным к нему. Благодаря этому потоки не конкурируют за доступ к объекту. Объект принадлежит потоку и не используется совместно. В Java желаемого эффекта можно добиться при помощи класса `ThreadLocal` (<http://mng.bz/aD8B>). Он позволяет обернуть системный компонент, который будет привязан к конкретному потоку. С точки зрения кода объект находится внутри экземпляра `ThreadLocal`, как показывает следующий листинг.

Листинг 1.6. Привязка к потоку с использованием ThreadLocal

```
private static ThreadLocal<SystemComponent> threadLocalValue = new
    ThreadLocal<>();

public static void set() {
    threadLocalValue.set(new SystemComponent());
}

public static void executeAction() {
    SystemComponent systemComponent = threadLocalValue.get();
}

public static SystemComponent get() {
    return threadLocalValue.get();
}
```

Логика привязки `SystemComponent` к конкретному потоку инкапсулируется в экземпляре `ThreadLocal`. Когда поток А вызывает метод `set()`, внутри `ThreadLocal` создается новый экземпляр `SystemComponent`. Важно, что этот экземпляр доступен только для этого потока. Если другой поток (скажем, В) вызовет `executeAction()` без предварительного вызова `set()`, он получит `null`-экземпляр `SystemComponent`, потому что для этого потока компонент еще не был создан вызовом `set()`. Новый экземпляр, предназначенный для этого потока, будет создан и станет доступен только после того, как поток В вызовет метод `set()`.

Происходящее можно упростить, передав поставщик (`supplier`) при вызове метода `withInitial()`. Он будет вызван, если потоково-локальный объект еще не получил значения, так что риск получить `null` отсутствует. В следующем листинге показана возможная реализация.

Листинг 1.7. Привязка к потоку с исходным значением

```
static ThreadLocal<SystemComponent> threadLocalValue =
    ThreadLocal.withInitial(SystemComponent::new);
```

Применение этого паттерна устраняет конкуренцию, что улучшает производительность. Обратной стороной такого решения становится усложнение системы.

ПРИМЕЧАНИЕ

Каждый раз, когда код на стороне вызова хочет обратиться к одиночному экземпляру, обращение не обязано использовать метод `getInstance()`. Можно обратиться к одиночному экземпляру один раз и присвоить его переменной (ссылке). После этого дальнейшие обращения могут получить доступ к одиночному объекту по ссылке без необходимости вызывать `getInstance()`. Такое решение снижает конкуренцию потоков.

Одиночный экземпляр также можно внедрить в другие компоненты, которые должны его использовать. В идеале приложение создает все компоненты в одном месте и помещает их в сервисы (например, с применением внедрения

зависимостей). В таком случае паттерн Одиночка может вообще не понадобиться. Вы просто создаете один экземпляр объекта, который должен находиться в общем доступе, и внедряете его во все зависимые сервисы (<http://mng.bz/g4dE>). Альтернатива этому — использование перечисляемого типа, в основе которого лежит паттерн Одиночка. Чтобы проверить предположения, измерим временные характеристики кода.

1.2.1. Измерение скорости выполнения

К этому моменту мы создали три потоково-безопасные реализации паттерна Одиночка:

- с синхронизацией для всех операций;
- с блокировкой с двойной проверкой;
- с привязкой к потокам (с использованием `ThreadLocal`).

Предполагается, что первая версия будет самой медленной, но данных об этом у нас пока нет. Создадим тест производительности, который будет проверять все три реализации. Воспользуемся инструментарием проверки производительности JMH (<https://openjdk.java.net/projects/code-tools/jmh/>), который мы еще применим в дальнейшем для проверки быстрей действия кода.

Создадим хронометражный тест, который выполняет 50 000 операций получения (одиночного) объекта `SystemComponent` (листинг 1.8). В нем будут реализованы три теста, использующих разные подходы к функциональности паттерна Одиночка. Чтобы проверить, как конкуренция потоков влияет на производительность, код будет выполняться в 100 одновременно работающих потоках. Результаты (среднее время выполнения) будут выведены в миллисекундах.

Листинг 1.8. Создание хронометражных тестов реализаций паттерна Одиночка

```
@Fork(1)
@Warmup(iterations = 1)
@Measurement(iterations = 1)
@BenchmarkMode(Mode.AverageTime)
@Threads(100)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class BenchmarkSingletonVsThreadLocal {
    private static final int NUMBER_OF_ITERATIONS = 50_000;

    @Benchmark
    public void singletonWithSynchronization(Blackhole blackhole) {
        for (int i = 0; i < NUMBER_OF_ITERATIONS; i++) {
            blackhole.consume(
                SystemComponentSingletonSynchronized.getInstance());
        }
    }
}
```

Код выполняется в 100 одновременно работающих потоках

В первом тесте используется `SystemComponent SingletonSynchronized`

```

@Benchmark
public void singletonWithDoubleCheckedLocking(Blackhole blackhole) {
    for (int i = 0; i < NUMBER_OF_ITERATIONS; i++) {
        blackhole.consume(
➔ SystemComponentSingletonDoubleCheckedLocking.getInstance());
    }
}

@Benchmark
public void singletonWithThreadLocal(Blackhole blackhole) {
    for (int i = 0; i < NUMBER_OF_ITERATIONS; i++) {
        blackhole.consume(SystemComponentThreadLocal.get());
    }
}

```

Тесты для SystemComponentSingletonDoubleCheckedLocking

Получить результаты для SystemComponentThreadLocal

Выполнив тест, мы получаем среднее время на 50 000 вызовов для 100 одновременных потоков. В конкретной среде числа могут быть другими, но общие тенденции останутся теми же, как показывает следующий листинг.

Листинг 1.9. Просмотр результатов хронометражного теста реализаций паттерна Одиночка

Benchmark	Mode	Cnt	Score	Error	Units
CH01.BenchmarkSingletonVsThreadLocal.singletonWithDoubleCheckedLocking	avgt		2.629		ms/op
CH01.BenchmarkSingletonVsThreadLocal.singletonWithSynchronization	avgt		316.619		ms/op
CH01.BenchmarkSingletonVsThreadLocal.singletonWithThreadLocal	avgt		5.622		ms/op

Из результатов видно, что реализация `singletonWithSynchronization` действительно оказалась самой медленной. Среднее время выполнения логики хронометража составило около 300 мс (миллисекунд). Затем идут два решения с более высокими результатами. Реализация `singletonWithDoubleCheckedLocking` показывает лучший результат (около 2,6 мс), а реализация `singletonWithThreadLocal` завершилась за 5,6 мс. Можно сделать вывод, что усовершенствование исходной версии паттерна Одиночка обеспечивает примерно 50-кратное повышение производительности для потоково-локального решения и 115-кратное для решения с блокировкой с двойной проверкой.

Проверяя предположения, можно принимать эффективные решения в многопоточном контексте. Если потребуется выбрать одно решение вместо другого при сравнимой производительности, можно отдать предпочтение более прямолинейному решению. Тем не менее без реальных данных трудно сделать полностью рациональный выбор.

Теперь рассмотрим компромиссы для архитектурных решений. В следующем разделе вы узнаете о микросервисных и монолитных архитектурах и о связанных с ними компромиссах.

1.3. АРХИТЕКТУРНЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ И ПОЧЕМУ ОНИ РАБОТАЮТ НЕ ВСЕГДА

Выше мы рассмотрели низкоуровневые паттерны программирования и компромиссы, приводящие к разным вариантам структуры кода. При всей их важности вам, скорее всего, не составит труда изменить эти низкоуровневые части, меняя контекст приложения. Вторая часть книги будет посвящена архитектурным паттернам проектирования: их изменять сложнее, поскольку они охватывают всю архитектуру нескольких сервисов, образующих систему. Пока мы сосредоточимся на микросервисной (см. <http://mng.bz/enlv>) архитектуре — одном из самых популярных паттернов создания современных программных систем.

Микросервисная архитектура имеет ряд преимуществ перед созданием одной монолитной системы, в которой реализована вся бизнес-логика. Тем не менее она подразумевает усложнение системы и значительные затраты на обслуживание. Рассмотрим важнейшие преимущества микросервисной архитектуры перед монолитной.

1.3.1. Масштабируемость и эластичность

Системы, которые мы создаем, должны справляться с высоким трафиком и при этом адаптироваться и масштабироваться в зависимости от потребностей. Если один узел приложения способен обрабатывать N запросов в секунду и на нем наблюдается выброс трафика, микросервисная архитектура позволит быстро масштабироваться по горизонтали (рис. 1.3). Конечно, приложение должно быть написано так, чтобы поддерживать возможность простого масштабирования. Кроме того, оно должно использовать имеющиеся компоненты.

Например, можно добавить новый экземпляр существующего микросервиса, чтобы система могла обрабатывать $\sim 2 \times N$ запросов в секунду (где 2 — количество сервисов, а N — количество запросов, которые может обрабатывать один сервис). Тем не менее этого показателя можно достигнуть только в том случае, если базовый уровень доступа к данным способен масштабироваться столь же эффективно.

Конечно, может существовать некий верхний порог масштабируемости, после которого добавление новых узлов не сильно увеличит пропускную способность. Он может зависеть от предела масштабируемости используемых компонентов — базы данных, очереди, пропускной способности сети и т. д.

Тем не менее добиться общей масштабируемости микросервисной архитектуры обычно проще по сравнению с монолитным подходом. Монолитные архитектуры не позволяют масштабироваться с нужной скоростью после достижения верхнего порога ресурсов.

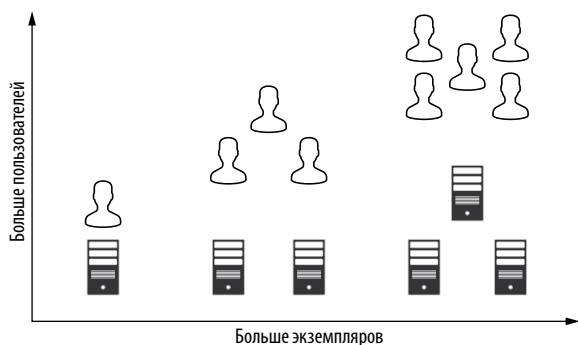


Рис. 1.3. Горизонтальное масштабирование означает добавление большего количества машин в пул ресурсов по мере возрастания потребности

Приложение также можно масштабировать вертикально за счет добавления большего количества процессоров, памяти или дискового пространства в вычислительный узел, и здесь также существует предел, при достижении которого масштабирование становится невозможным. Например, если монолитное приложение развернуто в облаке, его можно масштабировать за счет перехода на более мощный облачный вычислительный узел (с большим количеством процессоров или памяти). Такой подход отлично работает, пока есть возможность добавлять новые ресурсы. Однако есть риск, что провайдер облачного сервиса не сможет предложить более мощную машину для развертывания. В таком случае масштабирование по горизонтали окажется более гибким методом. Если приложение написано так, чтобы иметь возможность развертывания на N экземплярах, можно добавить в него новые экземпляры для повышения общей пропускной способности.

1.3.2. Скорость разработки

В микросервисной архитектуре задачи легко распределить между несколькими командами. Допустим, команда А работает над бизнес-функциональностью, которая будет реализована в отдельном микросервисе. В то же время команда В может сосредоточиться на другой части бизнес-области. Команды работают независимо друг от друга и эффективнее продвигаются вперед.

При работе с микросервисами отсутствует координация на уровне кодовой базы. Команды сами принимают решения относительно технологий и развиваются быстрее. Если к команде, задействованной в своей части бизнес-области, присоединяется новый участник, ему проще понять систему и приступить к работе.

Каждая команда может внедрить свою кодовую базу независимо, что делает процесс развертывания более надежным. В результате он происходит чаще

и с меньшим риском. Даже если команда случайно внесет баг, количество изменений в развертываемой версии будет меньше. Благодаря этому решение потенциальных проблем ускоряется. Трудности с отладкой могут появиться из-за ошибок, возникших вследствие интеграции двух слишком детализированных микросервисов. В таком случае необходимо обратиться к трассировке для отслеживания запросов, проходящих через несколько микросервисов (<http://mng.bz/p2w8>).

С другой стороны, в монолитных архитектурах кодовая база часто доступна разным командам. Если код приложения хранится в одном репозитории, а приложение достаточно сложное, несколько команд могут работать над ним одновременно. В таких ситуациях возрастает вероятность конфликтов в коде, и, как следствие, значительная часть рабочего времени может уйти на их разрешение. Конечно, если код продукта структурируется в модульной форме, этот эффект можно снизить. Тем не менее всегда будет необходимость в учащенной перебазирке, так как основная кодовая база продукта меняется быстрее, если над ней трудится много разработчиков. При сравнении монолитной архитектуры с микросервисной легко заметить, что код выделенной бизнес-области часто значительно короче. Это значит, что конфликтов, скорее всего, также будет меньше.

В монолитных приложениях развертывание выполняется реже. Причина в том, что больше функций объединяется в главную кодовую ветвь (потому что над ней работает много людей). Чем шире функционал, тем дольше он тестируется. Поскольку в одном выпуске развертывается множество функций, вероятность внесения бага в систему возрастает.

Стоит заметить, что число подобных проблем можно сократить, создав более надежный конвейер непрерывной интеграции (или непрерывного развертывания). Его можно запускать чаще, регулярно создавая новые версии приложения с меньшим количеством функций. Код свежего выпуска проще анализировать и отлаживать при обнаружении проблем. Чем меньше новых функций, тем быстрее вы разберетесь в причинах неполадок. Если сравнить этот подход с циклом выпуска, в котором новое приложение строится реже, очевидно, что в таком выпуске будет больше новых функций, внедряющихся одновременно. Чем шире функционал выпуска, тем больше в нем потенциальных проблем и тем сложнее его отладить.

1.3.3. Сложность микросервисов

Узнав о преимуществах микросервисной архитектуры перед монолитной, познанимся и с ее недостатками. Микросервисная архитектура — сложная конструкция с множеством подвижных частей. Чтобы обеспечить ее масштабируемость, необходимо иметь подходящий балансировщик нагрузки, который ведет список работающих сервисов и маршрутизирует трафик. Нижележащие сервисы могут масштабироваться с увеличением и уменьшением, то есть появляться и исчезать.

Отслеживание таких изменений — непростая задача. Для этого понадобится новый компонент — реестр сервисов (рис. 1.4).

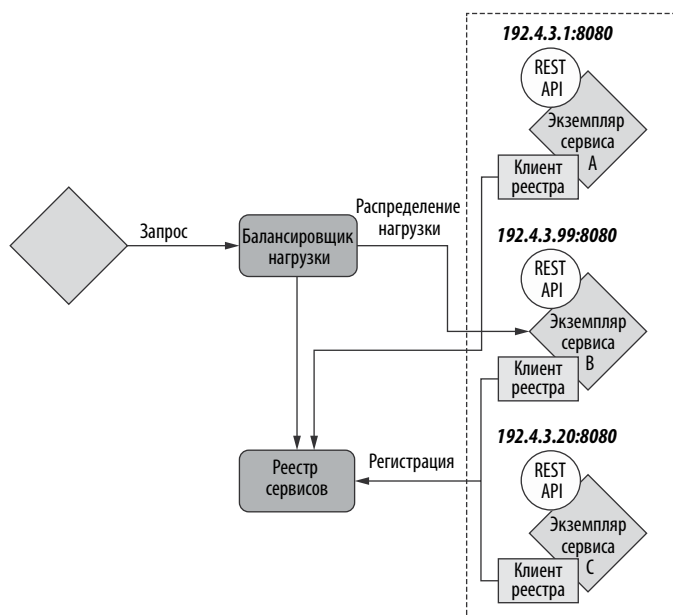


Рис. 1.4. Реестр микросервисов

Каждому микросервису необходим работающий клиент реестра, отвечающий за регистрацию его в реестре. После регистрации балансировщик нагрузки сможет маршрутизировать трафик новому экземпляру. Отмена регистрации обеспечивается реестром сервисов путем проверки состояния экземпляров сервисов. Это одна из сложностей данной архитектуры, значительно затрудняющая развертывание.

Разобравшись с сильными и слабыми сторонами задачи, вы должны учесть контекст, чтобы выбрать подходящее проектное решение. Если контекст показывает, что возможности масштабирования ограничены, и у вас небольшая команда разработчиков, решением может стать монолитная архитектура. Процессы оценки проектных решений, описанные в следующих главах, похожи на представленный здесь: определение плюсов и минусов каждого из вариантов, добавление контекста и выбор оптимального подхода для конкретной ситуации.

В этой главе были приведены примеры компромиссов проектирования, которые рассматриваются далее в нашей книге. Вы изучили низкоуровневые компромиссы, подразумевающие выбор соотношения модульных и интеграционных тестов в приложениях. Также вы узнали, что проверенные временем паттерны (такие,

как Одиночка) не всегда лучший выбор в зависимости от контекста использования. Они могут повлиять на производительность системы в многопоточных средах — например, из-за конкуренции потоков. Наконец, мы рассмотрели такие паттерны проектирования, как микросервисные и монолитные архитектуры, которые служат примером высокоуровневых проектных решений.

Следующая глава посвящена компромиссам между дублированием кода и повторным использованием. Вы убедитесь, что дублирование кода не всегда плохо — опять же в зависимости от контекста.

ИТОГИ

- Если время разработки продукта ограничено, необходимо учитывать этапы, следующие после проектирования (например, покрытие кода модульными и интеграционными тестами).
- Проверенные временем низкоуровневые паттерны проектирования (например, Одиночка) — не всегда лучшее решение (скажем, в отношении потоковой безопасности). Это зависит от контекста приложения.
- Высокоуровневые микросервисные архитектуры подходят не для каждой задачи; необходим фреймворк для оценки проектных решений.

2

Дублирование кода не всегда плохо: дублирование кода и гибкость

https://t.me/it_books

В этой главе:

- ✓ Совместное использование общего кода в независимых кодовых базах.
- ✓ Компромиссы между дублированием кода, гибкостью и доставкой результата.
- ✓ Когда дублирование кода становится грамотным решением, обеспечивающим слабую связанность.

Принцип DRY (Don't Repeat Yourself, «не повторяйтесь») — одно из самых известных правил программирования. В его основе лежит идея отказа от дублирования кода, что сокращает число багов и повышает потенциал повторного использования программного продукта. Однако фанатично следовать принципу DRY при построении любой системы опасно, поскольку такой подход скрывает многие сложности. Проще соблюдать принцип DRY, если создаваемая система монолитна; это означает, что почти вся кодовая база хранится в одном репозитории.

В наши дни разработчики часто включают в распределенные системы множество подвижных частей. В таких архитектурах решение об отказе от дублирования

кода приводит к другим компромиссам — например, созданию сильной связанности между компонентами или сокращению скорости разработки в команде. Если один фрагмент кода используется в нескольких местах, его изменение может потребовать значительной координации, а это замедляет создание коммерческой ценности. В этой главе подробно рассматриваются паттерны и компромиссы, относящиеся к дублированию кода. Мы попытаемся ответить на вопрос: когда дублирование кода может быть разумным компромиссом, а когда его следует избегать?

Начнем с наличия дублированного кода в двух кодовых базах. Затем сократим дублирование за счет использования общей библиотеки. После этого применим другой подход к извлечению общей функциональности — используем микросервис, инкапсулирующий это поведение. Далее рассмотрим наследование как паттерн для устранения дублирования в коде. Вы увидите, что такое решение тоже сопряжено с затратами, которые нельзя просто игнорировать.

2.1. ОБЩИЙ КОД В КОДОВЫХ БАЗАХ И ДУБЛИРОВАНИЕ

Для анализа первой задачи рассмотрим совместное использование кода в контексте микросервисной архитектуры. Представьте, что в проекте две команды. Команда А работает над сервисом платежей, а команда В — над сервисом личных данных. Сценарий изображен на рис. 2.1.

Сервис платежей предоставляет HTTP API с конечной точкой `/payment`. Сервис личных данных предоставляет свою бизнес-логику через конечную точку `/person`. Будем считать, что обе кодовые базы написаны на одном языке программирования. На этой стадии обе команды продвигаются в работе и могут быстро поставлять очередные версии продукта.

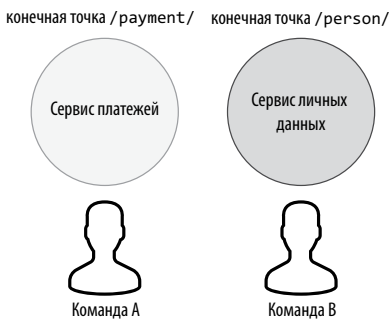


Рис. 2.1. Два независимых сервиса: платежей и личных данных

Один из важнейших факторов, обуславливающих высокую скорость разработки, — отсутствие необходимости синхронизации между командами. Применяя закон Амдала, можно даже вычислить, в какой степени синхронизация влияет на общее время производства продукта. Согласно формуле, чем меньше синхронизации (а следовательно, чем больше параллельная часть работы), тем выше эффект от подключения новых ресурсов к решению задачи. Этот принцип проиллюстрирован на рис. 2.2.

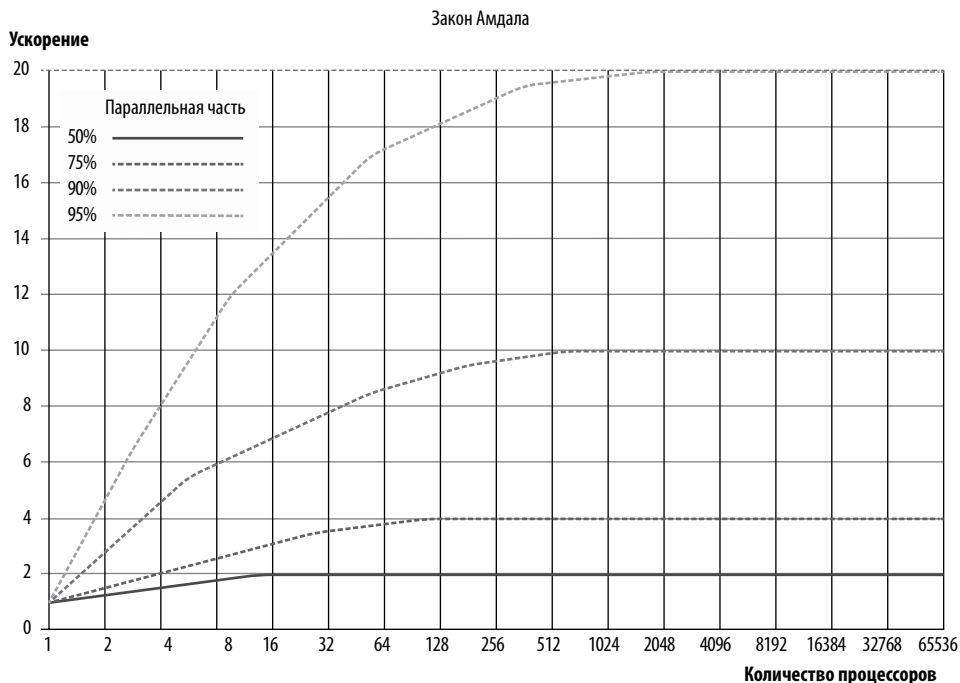


Рис. 2.2. Закон Амдала находит максимальное ожидаемое улучшение для всей системы в зависимости от доли распараллеливаемой работы

Например, если задача распараллеливается 50 % времени (а другие 50 % требует синхронизации), скорость обработки при добавлении ресурсов (количество процессоров на диаграмме) значительно не повысится. Но чем сильнее распараллелена задача и чем меньше затраты на синхронизацию, тем больший выигрыш в скорости достигается при добавлении новых ресурсов.

Формула Амдала может использоваться для вычисления распараллеливания одновременной обработки и выгоды от добавления новых ядер, но ее также можно адаптировать для участников команды, работающих над конкретной задачей (<http://mng.bz/OG4R>). Синхронизация, снижающая степень параллелизма, может

быть представлена временем, затрачиваемым на собрания, объединением задач и другими действиями, требующими присутствия всей команды.

Когда код дублируется, он разрабатывается обеими командами независимо и синхронизации между ними не требуется. Следовательно, добавление нового участника в команду приведет к повышению производительности. Ситуация меняется, когда дублирование кода устраняется и двум командам приходится работать над одним фрагментом кода и мешать друг другу.

2.1.1. Добавление нового бизнес-требования, для которого дублирование кода необходимо

Через какое-то время в процессе работы над сервисами возникает новое бизнес-требование: добавить авторизацию в оба HTTP API. Первое решение команд — реализовать компонент авторизации в обеих кодовых базах. На рис. 2.3 изображена обновленная архитектура.

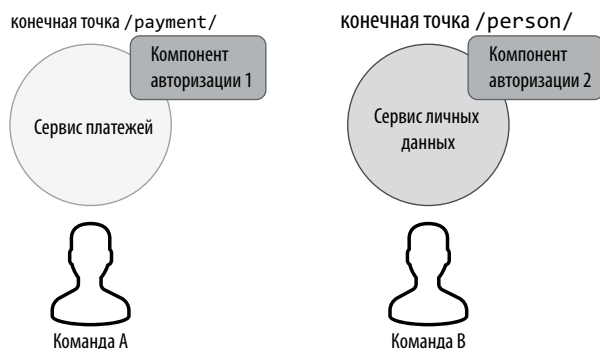


Рис. 2.3. Новый компонент авторизации

Обе команды разрабатывают и сопровождают похожие компоненты авторизации. Однако работа двух групп все еще остается независимой.

В этом сценарии следует помнить, что в примере используется упрощенная версия аутентификации на основе маркеров, но такое решение уязвимо для атак повторного воспроизведения (<http://mng.bz/YgYB>), поэтому оно не подходит для реальной эксплуатации. Упрощенная версия позволяет не отвлекаться от главных аспектов, обсуждаемых в этой главе. Заметим, что обеспечить безопасность достаточно сложно. Если команды работают независимо, то вероятность, что они обе с этим справятся, весьма низкая. Даже если на разработку общей библиотеки уйдет больше времени, выгода от предотвращения инцидентов безопасности может того стоить.

2.1.2. Реализация нового бизнес-требования

Рассмотрим сервис платежей. Он предоставляет конечную точку HTTP `/payment` и использует только один ресурс `@GET` для получения всех платежей от заданного маркера, как видно из следующего листинга.

Листинг 2.1. Реализация конечной точки `/payment`

```
@Path("/payment")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class PaymentResource {

    private final PaymentService paymentService = new PaymentService();
    private final AuthService authService = new AuthService();

    @GET
    @Path("/{token}")
    public Response getAllPayments(@PathParam("token") String token) {
        if (authService.isTokenValid(token)) {
            return Response.ok(paymentService.getAllPayments()).build();
        } else {
            return Response.status(Status.UNAUTHORIZED).build();
        }
    }
}
```

Предоставляет интерфейс для микросервиса платежей

Создает экземпляр AuthService

Проверяет маркер с использованием AuthService

Как видно из листинга 2.1, `AuthService` проверяет маркер, после чего вызывающая сторона переходит к платежному сервису, который возвращает все платежи. Реальная логика `AuthService` будет сложнее. Взгляните на упрощенную версию в следующем листинге.

Листинг 2.2. Создание сервиса авторизации

```
public class AuthService {

    public boolean isTokenValid(String token) {
        return token.equals("secret");
    }
}
```

ПРИМЕЧАНИЕ

На практике две команды вряд ли создадут полностью одинаковые интерфейсы, имена методов, сигнатуры и т. д. Одно из преимуществ раннего принятия решения о совместном использовании кода в том, что на расхождение двух реализаций в таком случае останется меньше времени.

Вторая команда работает над сервисом личных данных, который предоставляет конечную точку HTTP `/person`. Он также выполняет авторизацию на основе маркера, как показано в листинге 2.3.

Листинг 2.3. Реализация конечной точки /person

```

@Path("/person")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class PersonResource {

    private final PersonService personService = new PersonService();
    private final AuthService authService = new AuthService();

    @GET
    @Path("/{token}/{id}")
    public Response getPersonById(@PathParam("token") String token,
        @PathParam("id") String id) {
        if (authService.isTokenValid(token)) {
            return Response.ok(personService.getById(id)).build();
        } else {
            return Response.status(Status.UNAUTHORIZED).build();
        }
    }
}

```

Предоставляет интерфейс
HTTP для микросервиса
личных данных

Создает экземпляр
AuthService

Проверяет маркер
с использованием
AuthService

Сервис также интегрирует `AuthService`. Он проверяет маркер, предоставленный пользователем, после чего получает данные `Person` через `PersonService`.

2.1.3. Оценка результата

Так как в этой точке обе команды ведут разработку независимо друг от друга, код и работа дублируются.

- *Дублирование может привести к большему числу багов и ошибок.* Например, если команда `Person` исправит баг в своем компоненте авторизации, это не гарантирует, что команда `Payment` не допустит тот же баг.
- *Когда один и тот же или похожий код дублируется в независимых кодовых базах, программисты не обмениваются знаниями.* Например, команда `Person` находит баг в вычислениях с маркером и исправляет его в своей кодовой базе. К сожалению, эта правка автоматически не распространяется на кодовую базу `Payment`. Команде `Payment` придется исправить баг позже, независимо от команды `Person`.
- *Работа без координации может идти быстрее.* Но даже в этом случае обе команды выполняют значительный объем похожей работы.

На практике вы, вероятно, не станете реализовывать логику с нуля, а примените стратегии аутентификации, проверенные в условиях реальной эксплуатации, такие как OAuth (<https://oauth.net/2/>) или JWT (<https://jwt.io/>). Эти стратегии еще более эффективны в контексте микросервисной архитектуры. Оба способа дают множество преимуществ, когда несколько сервисов требуют аутентификации для обращения к ресурсам других сервисов. Мы не будем рассматривать конкретную стратегию аутентификации или авторизации. Вместо этого сосредоточимся на

таких аспектах кода, как гибкость, удобство обслуживания и сложность. В следующем разделе вы увидите, как проблема дублирования решается выделением общего кода в совместно используемую библиотеку.

2.2. БИБЛИОТЕКИ И СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ КОДА В КОДОВЫХ БАЗАХ

Предположим, из-за того что значительная часть кода дублируется в двух независимых кодовых базах, обе команды решают выделить общий код в отдельную библиотеку. Выделим код сервиса авторизации в специальный репозиторий. Одной команде потребовалось создать процесс развертывания для новой библиотеки. Самый распространенный сценарий — публикация библиотеки во внешнем менеджере репозиториях, таком как JFrog Artifactory (<https://jfrog.com/open-source/>). Этот сценарий показан на рис. 2.4.

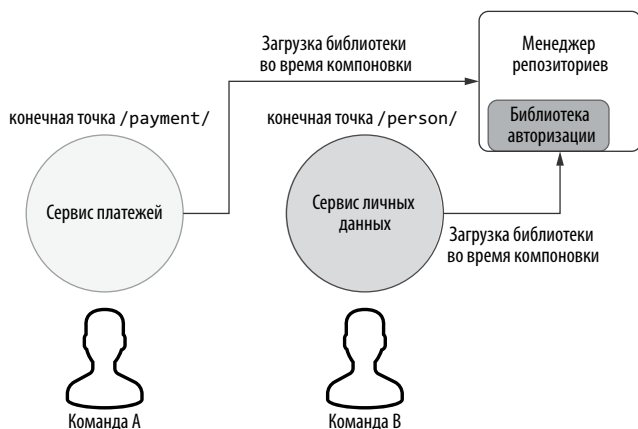


Рис. 2.4. Загрузка общей библиотеки из менеджера репозитория

После того как общий код попадает в менеджер репозиториях, оба сервиса могут загрузить библиотеку во время компоновки и использовать классы, поставляемые с ней. Применяя такой подход, можно исключить любое дублирование кода за счет его хранения в одном месте.

Одно из очевидных преимуществ исключения дублирования — общее качество кода. Хранение совместно используемой библиотеки позволяет обеим командам взаимодействовать друг с другом и повышает качество кодовой базы. Благодаря этому исправления багов немедленно распространяются на все клиенты библиотеки, так что работа не дублируется. Рассмотрим недостатки и компромиссы, на которые приходится идти при выборе этого подхода.

2.2.1. Оценка компромиссов и недостатков совместно используемых библиотек

Как только мы выделяем новую библиотеку, она становится новой сущностью с собственным стилем программирования, процессом развертывания и практикой кодирования. В этом контексте под библиотекой понимается код, упакованный в некотором формате (JAR, DLL или *.so на платформах Linux и т. д.), который может использоваться в разных проектах. Команда (или независимый разработчик) должна принять ответственность за новую кодовую базу. Кто-то должен настраивать процесс развертывания, проверять качество кода, разрабатывать новую функциональность и т. д. Впрочем, это можно отнести к второстепенным фиксированным затратам.

При выборе совместно используемых библиотек нужно разработать сопутствующие процессы, включая практики кодирования, развертывания и т. д. Но процесс создается один раз, а применять его можно многократно. Затраты на добавление первой библиотеки могут быть значительными, но затраты на добавление второй будут гораздо меньше.

Один из самых очевидных компромиссов такого подхода в том, что язык, на котором создается новая библиотека, должен совпадать с языком клиентов, которые ее используют. К примеру, если сервисы платежей и личных данных разрабатываются на разных языках (скажем, на Python или Java), создать новую библиотеку не удастся. Однако в реальности это не проблема, потому что сервисы создаются на одном языке или семействе языков (например, JVM).

Можно создать экосистему сервисов, где они пишутся с использованием разных технологий. Однако это значительно усложнит систему в целом. Это также означает, что вам потребуются люди с опытом в целом ряде технологий. Кроме того, придется задействовать множество инструментов — скажем, системы сборки или пакетные менеджеры из различных технологических стеков. В зависимости от выбранного языка придется иметь дело с его особенной экосистемой.

УЧАСТИЕ В РАЗРАБОТКЕ С ОТКРЫТЫМ КОДОМ

В экосистеме JVM существует активное сообщество разработки с открытым кодом, которое занимается созданием и обслуживанием различных библиотек. Прежде чем принимать решение о выделении отдельной библиотеки, следует провести исследование и узнать, существует ли готовая библиотека с открытым кодом, решающая актуальную задачу. Впрочем, возможно, ее придется адаптировать или немного расширить под насущные потребности.

Можно также сделать свой код открытым, если похожих библиотек не существует. Участвуя в проекте с открытым кодом, вы предоставляете свой код другим потенциальным пользователям. Кроме того, вам бесплатно достается процесс развертывания и реклама. Скорее всего, другие разработчики также найдут вашу библиотеку и используют ваш код повторно.

Зачастую можно писать библиотеку на другом языке (скажем, C) и упаковывать ее в язык платформенного интерфейса (например, Java Native Interface) по своему выбору. Однако это может создать проблемы, потому что коду понадобится еще один уровень перенаправления. Может оказаться, что код, упакованный в платформенный интерфейс, не портируется между операционными системами, а вызовы его методов выполняются медленнее (по сравнению с вызовами методов Java). Поэтому будем считать, что рассматриваемые далее экосистемы имеют одинаковый технологический стек.

Информация о новой библиотеке должна распространяться в компании, чтобы другие команды могли узнать о ней и использовать в случае необходимости. Иначе мы получим гибридный подход: одни команды будут применять новую библиотеку, а другие продолжают дублировать код.

Менеджер репозитория — подходящее место для совместно используемой библиотеки, но придется поддерживать качественную документацию. Обычно хорошее покрытие тестами помогает другим разработчикам вносить вклад в библиотеку. Если у вас есть набор тестов, которым могут пользоваться и с которым могут экспериментировать другие разработчики, к вашей библиотеке будут обращаться чаще и больше. Также отметим, что документация иногда устаревает. Следовательно, ее необходимо регулярно обновлять.

Тесты, как и код, требуют регулярного обслуживания и обновления, что помогает эффективно продвигать библиотеку в компании и формирует у потенциальных пользователей уверенность в ее общем качестве. Конечно, если вы выбрали подход с дублированием, придется тестировать дублируемый код во всех местах. А это значит, что может появиться дублируемый тестовый код.

Хорошее тестовое покрытие — это не повод игнорировать обновление документации. Непросто научиться работать с новой библиотекой, лишь изучая ее тесты, если только они не написаны специально для этой цели. Тесты должны покрывать все способы использования библиотеки, а не только предпочтительные. Они могут помочь с поиском ответа на конкретные вопросы, но они не настолько наглядны, как специальная страница с учебными примерами и руководством по началу работы.

2.2.2. Создание совместно используемой библиотеки

При создании библиотеки следует стремиться к простоте. Это особенно важно, когда вы зависите от сторонней библиотеки. Предположим, что компонент авторизации должен зависеть от популярной библиотеки Java Google Guava (<https://github.com/google/guava>), поэтому вы явно объявляете эту зависимость. Когда платежный сервис импортирует новую библиотеку авторизации, он также получает транзитивную зависимость от Google Guava. Все отлично работает, пока платежному сервису не понадобится импортировать другую стороннюю библиотеку с прямой зависимостью от Google Guava, но другой версии. Этот сценарий изображен на рис. 2.5.

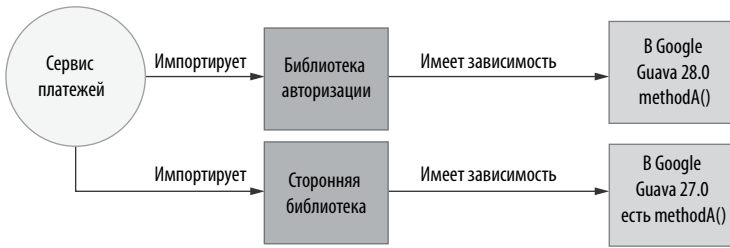


Рис. 2.5. Транзитивные зависимости, необходимые для реализации сервиса платежей

В такой ситуации платежный сервис будет содержать две версии одной библиотеки. Ситуация только усложняется, если основная версия библиотеки изменилась, — в этом случае двойная совместимость двух версий возможна, но не гарантирована. Более того, если обе библиотеки присутствуют в пути к классам, система сборки (например, Maven или Gradle) часто автоматически берет более новую версию, если обратное поведение не настроено в конфигурации. Например, возможна ситуация, в которой код сторонней библиотеки полагается на более старую версию Guava и вызывает метод `methodA()`, отсутствующий в новой версии. Если конфигурация не указывает, какую версию использовать, система сборки может выбрать новую версию библиотеки. В таком случае можно получить исключение `MethodNotFound` или нечто похожее. Дело в том, что сторонняя библиотека ожидает Guava 27.0 с вызываемым методом `methodA()`, а система сборки загрузила Guava 28, и сторонней библиотеке приходится использовать именно ее. Так возникают упомянутые выше проблемы.

Подобные конфликты сложно устранить, и они могут отбить желание других команд использовать ваш код, выделенный в библиотеку. А значит, ваша библиотека должна иметь как можно меньше прямых зависимостей. Эта тема более подробно рассматривается в главах 9 и 12 — в них основное внимание уделяется решениям, которые принимаются при выборе библиотек для создаваемых систем.

В этом сценарии предполагается, что созданная библиотека будет использоваться как в платежном сервисе, так и в сервисе личных данных. На этой стадии над самим сервисом авторизации не работает отдельная команда, так что обе они будут участвовать в разработке новой библиотеки. Это потребует планирования и координации между участниками обеих команд.

2.3. ВЫДЕЛЕНИЕ КОДА В ОТДЕЛЬНЫЙ МИКРОСЕРВИС

Совместное использование кода за счет библиотек может быть хорошим стартом, но, как видно из раздела 2.2.1, у такого решения есть несколько недостатков. Во-первых, разработчики, создающие библиотеку, должны учитывать совместимость и множество других факторов. Они не могут свободно пользоваться сторонними

библиотеками. Кроме того, импорт библиотеки означает, что между кодом продукта и библиотекой существует сильная связанность на уровне зависимостей. Это не означает, что она отсутствует в микросервисной архитектуре; сервисы могут связываться на уровне API, в форматах запросов и т. д. Связанность в библиотеке образуется в другом месте, нежели в микросервисной архитектуре.

Если дублируемая функциональность представляется отдельной бизнес-областью, мы можем задуматься о создании еще одного микросервиса, который обеспечивает ее через HTTP API. Например, можно определить отдельную бизнес-область, которая извлекает и предоставляет функции, изначально реализованные в другом месте. Наш компонент авторизации отлично для этого подходит, поскольку представляет ортогональный функционал проверки маркеров, а сервис авторизации имеет собственную бизнес-область. Можно найти бизнес-сущность, которую будет обрабатывать новый сервис, — например, сущность пользователя с логином и паролем.

ПРИМЕЧАНИЕ

Приведенный пример немного упрощен, но часто логике авторизации требуется доступ к другой информации (например, в базе данных). В таком случае, если разрешения хранятся, скажем, в базе данных, выделение логики в отдельные микросервисы еще более оправданно. Для простоты примера авторизация не требует доступа к внешнему сервису.

Добавление новых сервисов требует заметных усилий не только в разработке, но и в обслуживании. Очевидно, что сервис авторизации имеет собственную бизнес-область с отдельной бизнес-моделью. Функциональность аутентификации ортогональна существующей платформе. Сервисы платежей и личных данных не имеют прямого отношения к функциональности аутентификации. Учитывая сказанное, рассмотрим способы реализации сервиса авторизации. На рис. 2.6 показаны отношения между тремя сервисами.

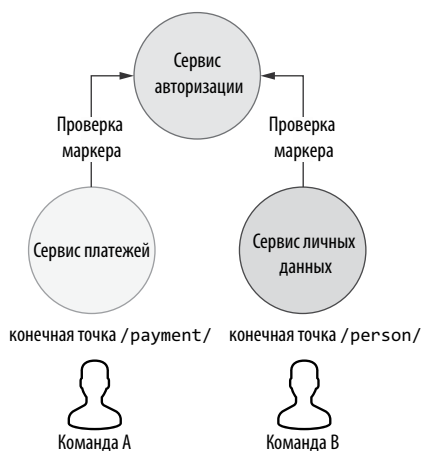


Рис. 2.6. Отношения сервиса авторизации с сервисами платежей и личных данных

Новая архитектура (рис. 2.6) содержит три отдельных микросервиса, соединяющихся друг с другом посредством HTTP API. А значит, сервисы личных данных и платежей должны выполнить один дополнительный запрос для проверки своих маркеров. Если у приложения невысокие требования к производительности, еще один вызов HTTP не создаст проблем (предполагается, что запрос выполняется внутри кластера или *замкнутой* сети и не передается случайному веб-серверу на другом конце земли). При этом логика сервиса авторизации, которая ранее дублировалась или выделялась в библиотеку, абстрагируется с использованием HTTP API, доступного через конечную точку `/auth`. Клиенты отправляют запросы для проверки этого маркера; при неудаче возвращается код ответа HTTP с признаком отсутствия авторизации (401). Если маркер действителен, то HTTP API возвращает код статуса ОК (200). Ниже приведен новый сервис авторизации.

Листинг 2.4. Конечная точка сервиса авторизации HTTP

```
@Path("/auth")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class AuthResource {

    private final AuthService authService = new AuthService();

    @GET
    @Path("/validate/{token}")
    public Response getAllPayments(@PathParam("token") String token) {
        if (authService.isTokenValid(token)) {
            return Response.ok().build();
        } else {
            return Response.status(Status.UNAUTHORIZED).build();
        }
    }
}
```

Так как AuthService все еще инкапсулирует логику проверки маркеров, вместо вызова библиотечной функции теперь выполняются запросы HTTP. Код находится в выделенном репозитории микросервиса авторизации. Сервисам платежей и личных данных теперь не нужно ни импортировать авторизацию напрямую, ни реализовывать их логику в своих кодовых базах. Им требуется лишь клиент HTTP, который отсылает запрос HTTP конечной точке `/auth` для проверки маркера. Ниже приведен код отправки запроса.

Листинг 2.5. Отправка запроса HTTP сервису AuthorizationService

```
// Отправка запроса отдельному сервису
public boolean isTokenValid(String token) throws IOException {
    CloseableHttpClient client = HttpClients.createDefault();
    HttpGet httpGet = new HttpGet("http:// /auth-service/auth/validate/" +
token);
    CloseableHttpResponse response = client.execute(httpGet);
    return response.getStatusLine().getStatusCode() == HttpStatus.SC_OK;
}
Отправляет запрос HTTP внешнему
сервису авторизации
```


В листинге 2.5 создается клиент HTTP, выполняющий запросы HTTP. В реальных системах клиент будет совместно использоваться в вызовах и компонентах, чтобы сократить количество открытых подключений и потребление ресурсов.

HttpClient (<https://hc.apache.org/>) выполняет запрос HTTP GET для проверки маркера. Если строка статуса ответа содержит код статуса ОК, это означает, что маркер действителен; в противном случае он недействителен.

ПРИМЕЧАНИЕ

Для предоставления доступа к сервису авторизации можно воспользоваться системой доменных имен (DNS) auth-service. Также возможно использование других механизмов обнаружения сервисов, таких как Eureka (<https://github.com/Netflix/eureka>), Consul (<https://www.consul.io/>) и т. д. Доступ к auth-service также возможен по статическому IP-адресу.

2.3.1. Компромиссы и недостатки отдельного сервиса

Отдельный микросервис решает некоторые проблемы, с которыми мы столкнулись при выделении общего кода в отдельную библиотеку. Решение с выделением отдельной библиотеки требует определенного изменения менталитета команды, использующей код. Когда вы импортируете библиотеку в свою кодовую базу, этот код становится вашим и вы несете за него ответственность. Кроме того, использование такой библиотеки подразумевает более сильную связанность, чем применение отдельного микросервиса.

При интеграции с другими микросервисами можно рассматривать их как «черный ящик». Единственной точкой интеграции является API, которым может быть HTTP или другой протокол. Теоретически к библиотеке можно относиться аналогично. К сожалению, как было показано в разделе 2.2, на практике библиотеку нельзя рассматривать как «черный ящик» из-за зависимостей, которые она вводит в код.

Вызов микросервиса означает, что необходимо также добавить новую зависимость в клиентскую библиотеку, используемую для выполнения кода. Теоретически это может привести к появлению проблемы транзитивных зависимостей, описанной в предыдущем разделе. И снова на практике большинство микросервисов должно использовать клиентскую библиотеку для вызовов других сервисов. Это может быть клиент HTTP или что-то другое в зависимости от используемого протокола. А значит, когда потребуется вызвать микросервисы из сервиса, вы с большой вероятностью будете использовать тот же клиент HTTP. Поэтому проблемы дополнительных зависимостей для каждого вызванного сервиса не существует.

Рассмотрим сервис авторизации как отдельный микросервис с собственным API. Вы уже видели, что такой подход решает некоторые проблемы использования

библиотеки. Но, с другой стороны, обслуживание отдельного микросервиса требует значительных усилий. Придется сделать намного больше, чем просто запрограммировать сервис и логику авторизации.

Наличие отдельного микросервиса означает, что потребуются создать процесс развертывания, который будет публиковать код в облачной или локальной инфраструктуре. Библиотеке также необходим процесс развертывания, но намного более прямолинейный: достаточно упаковать файл JAR и развернуть его в менеджере репозитория. Кто-то должен будет отслеживать работоспособность сервиса и реагировать при возникновении проблем. Следует учесть, что затраты на создание процесса развертывания, обслуживания, отслеживания и т. д. представляют собой значительные начальные расходы. Но когда процесс будет налажен, разработка последующих микросервисов станет проще и быстрее (аналогичные начальные расходы потребуются и для библиотек). Рассмотрим несколько важнейших факторов, которые нужно учитывать при выборе подхода с отдельным сервисом.

Процесс развертывания

Микросервис необходимо развернуть и запустить как процесс. Это означает, что его необходимо отслеживать, а команда должна реагировать на возможные проблемы или сбои. Поэтому при выделении отдельного микросервиса необходимо учитывать такие факторы, как создание, отслеживание и оповещение. Если в компании существует экосистема микросервисов, то решения по отслеживанию и оповещению, вероятно, уже настроены. Но если вы стали одним из первых, кто хочет использовать такую архитектуру, вам придется подготовить эти решения самостоятельно. Подготовка означает большое количество точек интеграции и, следовательно, значительный объем работы.

Управление версиями

Пожалуй, управлять версиями микросервиса несколько проще, чем версионировать библиотеку (в некоторых отношениях). Библиотека должна обеспечивать семантическую версионность, и при этом совместимость API в основных версиях не должна нарушаться. Управление версиями API микросервиса также должно соответствовать рекомендациям по сохранению обратной совместимости. Тем не менее на практике проще отслеживать использование конечных точек и быстро исключать их, если они больше не задействованы. Если вы разрабатываете библиотеку и переход на новую основную версию невозможен, будьте осторожны и не нарушайте совместимость. Нарушение будет означать, что клиенты после обновления версии библиотеки не будут компилироваться. Такого быть не должно.

При наличии HTTP API для количественного измерения каждой конечной точки можно воспользоваться простым счетчиком с использованием библиотеки

метрик — например, Dropwizard (<https://metrics.dropwizard.io/4.1.2/>). Если счетчик конкретной конечной точки не увеличивается в течение долгого времени, а сервис используется только внутри компании, стоит подумать об отказе от поддержки такой точки. Если конечная точка открыта и задокументирована, возможно, ее придется поддерживать дольше, ведь нельзя исключить, что кто-то начнет ею пользоваться. Поэтому даже когда метрика конкретной точки не растет, это не значит, что ее можно удалить.

Итак, мы видим, что микросервисы обеспечивают большую гибкость в отношении эволюции API. Совместимость более подробно рассматривается в главе 12.

Потребление ресурсов

Использование библиотеки клиентом означает, что объем вычислений и потребление ресурсов в клиентском коде может увеличиться. Для каждого запроса, выполняемого платежным сервисом, маркер проверки должен обрабатываться в коде. Если код потребляет много ресурсов, придется увеличить количество процессоров или объем памяти в зависимости от нагрузки.

Если логика проверки скрыта за API, который предоставляется отдельным сервисом, потребление ресурсов и масштабирование перестают быть прямой проблемой для клиента. Обработка будет выполняться конкретным экземпляром микросервиса. Если запросов слишком много, то команда, ответственная за сервис, должна отреагировать и масштабировать сервис в соответствии с нагрузкой.

Следует понимать, что клиентскому коду в таком случае потребуется дополнительный вызов HTTP, потому что каждое подтверждение должно пройти путь к микросервису и обратно. Если логика, скрытая за API микросервиса, достаточно тривиальна, может оказаться, что дополнительные затраты на вызов HTTP превышают затраты на выполнение логики на стороне клиента. Если логика более сложна, то затраты HTTP могут быть незначительны по сравнению с работой микросервиса. При принятии решения о том, следует ли выделять функционал во внешний сервис, необходимо учитывать этот компромисс.

Производительность

Наконец, необходимо вычислить, как выполнение дополнительных запросов HTTP влияет на производительность. С маркерами, используемыми для авторизации, обычно связывается определенный срок действия. Следовательно, их можно кэшировать для сокращения количества запросов, которые должен выдавать сервис. Для кэширования функционала придется использовать библиотеку кэширования в клиентском коде.

Часто оба подхода (библиотеки и внешние сервисы) используются для предоставления бизнес-функциональности. При выделении логики в отдельный

микросервис нужно выполнять дополнительный вызов HTTP для каждого запроса пользователя к сервису, что может быть серьезным недостатком. Необходимо вычислить, как это повлияет на задержку ответа и соглашения об уровне обслуживания (SLA) сервиса. Один из таких сценариев представлен на рис. 2.7.

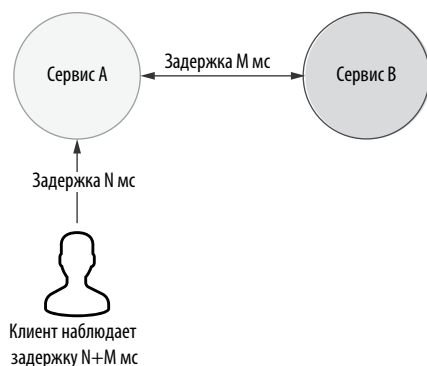


Рис. 2.7. Дополнительная задержка может повлиять на работу сервиса

Если, например, согласно SLA задержка 99-го процентиля должна быть менее n миллисекунд, добавление вызовов в другие микросервисы может нарушить SLA. Но если задержка 99-го процентиля меньше n , можно скрыть дополнительный вызов HTTP, выполняя некоторые запросы параллельно, за счет повторных попыток или упреждающего исполнения. Ситуация ухудшится, если задержка 99-го процентиля второго микросервиса больше n . В таком случае выполнить SLA не получится. Придется повысить задержку в требованиях SLA. Если это невозможно, уделите больше времени сокращению 99-го процентиля второго сервиса или используйте подход с выделением библиотеки.

Если задержка не критична, все равно следует учесть возможность каскадных сбоев (<http://mng.bz/GGrv>) и защититься от временной недоступности зависимого микросервиса. Проблема каскадных сбоев присуща не только микросервисам; она может произойти в любой внешней системе, к которой понадобится обращаться с вызовами (базе данных, API аутентификации и т. д.).

Если бизнес-процесс требует дополнительного внешнего запроса, необходимо решить, что делать, когда сервис недоступен. Можно реализовать повторную попытку с экспоненциальной отсрочкой, чтобы нисходящий сервис смог вернуться в рабочее состояние без перегрузки сервиса запросами. Используя этот прием, можно проверять состояние нисходящего сервиса каждые x миллисекунд, а когда он восстановится — постепенно наращивать трафик. При добавлении поведения экспоненциальной отсрочки повторные попытки должны выполняться реже: например, первая — через 1 секунду, вторая — через 10 секунд, третья — через

30 секунд и т. д. Если это не помогает и сервис недоступен в течение долгого времени, необходимо защититься от такой возможности при помощи паттерна Прерыватель (Circuit Breaker) (<https://martinfowler.com/bliki/CircuitBreaker.html>).

Следует предусмотреть альтернативное поведение, которое будет выполняться при недоступности нисходящей системы. Например, если у вас имеется платежная система и провайдер платежей недоступен, можно подтверждать оплату и списывать средства со счета через какое-то время только после того, как нисходящая система вернется в рабочее состояние. Такой подход необходимо реализовывать очень осторожно, и это должно быть осознанным бизнес-решением.

Обслуживание

Как видим, с выделением микросервисов связаны многочисленные компромиссы. В реальной жизни такой подход требует тщательного планирования и обслуживания. Лучше сравнить его с более прямолинейным методом выделения совместно используемых библиотек и перечислить все плюсы и минусы. Если логика, которую вы хотите выделить для совместного использования, проста и не имеет многочисленных зависимостей, возможно, стоит выделить библиотеку. С другой стороны, если логика сложна и может быть выделена как отдельный бизнес-компонент, можно подумать о создании нового микросервиса. Нужно учесть, что второй подход требует большего объема работы и, возможно, профильной команды, способной поддерживать работоспособность процесса.

2.3.2. Выводы о выделении отдельных сервисов

Рассмотрев все компромиссы, присущие микросервисам, мы видим, что у них много недостатков. Необходимо реализовать много новых частей. Даже если все сделано правильно, сбоев запросов, выполняющих внешние вызовы по сети (которая может быть ненадежной), все равно не избежать. Выбирая между решениями с библиотекой и микросервисами, необходимо учитывать все эти плюсы и минусы.

ПРИМЕЧАНИЕ

Абстрагирование функциональности в отдельный сервис или библиотеку проще передать на аутсорс. Например, делегировать реализацию логики аутентификации сторонней компании-разработчику. Впрочем, у такого подхода много недостатков, таких как (возможно) высокая цена, проблемы с координацией, отсутствие гибкости к изменениям и многое другое.

В следующем разделе мы проанализируем дублирование на более низком уровне. Вы увидите, как оно способствует слабой связанности.

2.4. УЛУЧШЕНИЕ СЛАБОЙ СВЯЗАННОСТИ ЗА СЧЕТ ДУБЛИРОВАНИЯ КОДА

В этом разделе проблема дублирования рассматривается на уровне кода. Конкретно — на структуре двух обработчиков запросов, имеющих дело с двумя разновидностями запросов трассировки.

Допустим, система должна обрабатывать два типа запросов: стандартные запросы трассировки и запросы на трассировку графа. Запросы могут поступать от разных API, использовать разные протоколы и т. д. По этой причине в программе будут два пути, обрабатывающих оба типа запросов независимо.

Начнем с более прямолинейного подхода с двумя разными компонентами обработчиков. `GraphTraceHandler` обрабатывает запросы на трассировку графа, а `TraceHandler` — нормальные запросы трассировки. Структура изображена на рис. 2.8.

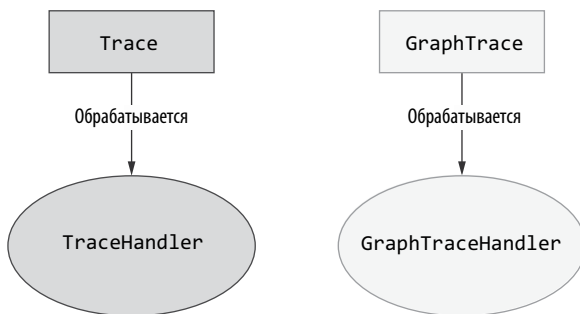


Рис. 2.8. Два независимых обработчика для выполнения запросов трассировки

Логика изолирована, связанность между двумя обработчиками отсутствует. Объекты `Trace` и `GraphTrace` похожи: они содержат информацию, если трассировка включена, и оба получают реальные данные. Для класса `GraphTrace` информация имеет тип `int`, тогда как для класса `Trace` это тип `String`, как видно из следующего листинга.

Листинг 2.6. Несвязанные классы `Trace` и `GraphTrace`

```

public class Trace {
    private final boolean isTraceEnabled;
    private final String data;

    public Trace(boolean isTraceEnabled, String data) {
        this.isTraceEnabled = isTraceEnabled;
        this.data = data;
    }
}
  
```

Задаёт тип данных для Trace

```

public boolean isTraceEnabled() {
    return isTraceEnabled;
}

public String getData() {
    return data;
}
}

public class GraphTrace {
    private final boolean isTraceEnabled;
    private final int data;

    public GraphTrace(boolean isTraceEnabled, int data) {
        this.isTraceEnabled = isTraceEnabled;
        this.data = data;
    }
    public boolean isTraceEnabled() {
        return isTraceEnabled;
    }

    public int getData() {
        return data;
    }
}

```

Обратите внимание: типы данных для GraphTrace и Trace различны

На первый взгляд классы похожи, но у них нет общей структуры. Они полностью отделены друг от друга.

Рассмотрим обработчики для запросов трассировки. Начнем с обработчика `TraceRequestHandler`. Он отвечает за буферизацию входящих запросов. На рис. 2.9 изображена схема работы `TraceRequestHandler`.

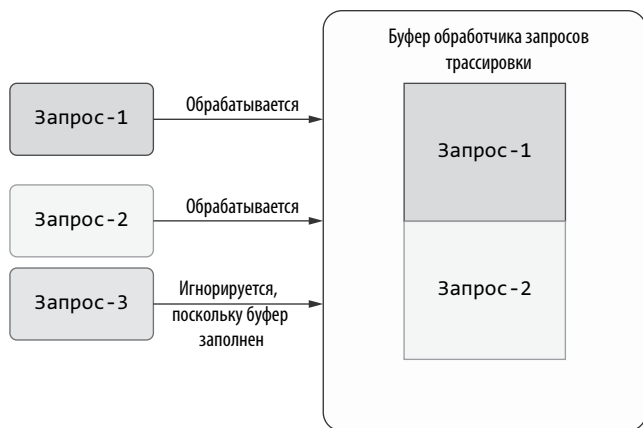


Рис. 2.9. Обработчик `TraceRequestHandler` буферизует входящие запросы

Как видно из диаграммы, `TraceRequestHandler` буферизует запросы, пока в буфере остается свободное место. Когда буфер заполняется, запрос (Запрос-3 на рис. 2.9) игнорируется.

Параметр `bufferSize` ограничивает размер буфера, переданный конструктору обработчика; он определяет, сколько элементов `TraceRequestHandler` сможет обработать. Запросы буферизуются в структуре данных списка. При заполнении буфера флагу `processed` присваивается значение `true`. В следующем листинге приведен код выделения обработчика.

Листинг 2.7. Выделенный обработчик `TraceRequestHandler`

```
public class TraceRequestHandler {
    private final int bufferSize;
    private boolean processed = false;
    List<String> buffer = new ArrayList<>();

    public TraceRequestHandler(int bufferSize) {
        this.bufferSize = bufferSize;
    }

    public void processRequest(Trace trace) {
        if (!processed && !trace.isTraceEnabled()) {
            return;
        }
        if (buffer.size() < bufferSize) {
            buffer.add(createPayload(trace));
        }

        if (buffer.size() == bufferSize) {
            processed = true;
        }

        private String createPayload(Trace trace) {
            return trace.getData() + "-content";
        }

        public boolean isProcessed() {
            return processed;
        }
    }
}
```

← Если размер буфера меньше `bufferSize`, присоединить к нему

← Если буфер заполнен, присвоить флагу `processed` значение `true`

Обратите внимание на метод `createPayload()`. Он содержит единственную логику, специфическую для класса `Trace`. Он получает запрос на трассировку, извлекает данные и создает строку, которая присоединяется к буферу.

Чтобы понять этот компонент, рассмотрим модульный тест, который будет обрабатывать пять запросов. При этом лимит буфера установлен равным 4. В таком случае после получения буфером четырех запросов пятый присоединен не

будет. В следующем листинге создается новый обработчик `TraceRequestHandler` с буфером размера 4 для реализации этой стратегии.

Последний запрос (со значением `e`) будет проигнорирован, потому что он выходит за пределы буфера.

Листинг 2.8. Создание модульного теста для `TraceRequestHandler`

```
@Test
public void shouldBufferTraceRequest() {
    // Дано
    TraceRequestHandler traceRequestHandler = new TraceRequestHandler(4);

    // Когда
    traceRequestHandler.processRequest(new Trace(true, "a"));
    traceRequestHandler.processRequest(new Trace(true, "b"));
    traceRequestHandler.processRequest(new Trace(true, "c"));
    traceRequestHandler.processRequest(new Trace(true, "d"));
    traceRequestHandler.processRequest(new Trace(true, "e"));

    // To
    assertThat(traceRequestHandler.buffer)
        .containsOnly("a-content", "b-content",
    ➔ "c-content", "d-content");
    assertThat(traceRequestHandler.isProcessed()).isTrue();
}
```

В содержимое буфера не входит элемент `e-content`

После обработки флаг `isProcessed` должен возвращать `true`

Как видим, буфер содержит только четыре записи. Чтобы понять, почему между обработчиками существует дублирование, необходимо проанализировать код `GraphTraceRequestHandler`. Собственно, у обработчиков трассировки графа и обычных обработчиков трассировки различается только метод `createPayload()`, который будет реализован в следующем листинге.

`graphTrace` извлекает данные и присоединяет к ним суффикс `nodeId`.

Листинг 2.9. Создание полезных данных для `GraphTraceRequestHandler`

```
private String createPayload(GraphTrace graphTrace) {
    return graphTrace.getData() + "-nodeId";
}
```

Остальной код обработки совпадает для двух компонентов. В этой точке становится ясно, что запросы трассировки и обработчики очень похожи. Они независимы и слабо связаны, но метод `processRequest()` для `TraceRequestHandler` достаточно сложен, и разработка этой логики в двух местах кода повышает риск ошибок и усложняет обслуживание.

Мы узнали о коде достаточно и можем решить, что общая логика должна быть выделена в отдельный родительский класс, а оба обработчика могут наследовать самые сложные части. В следующем разделе мы проанализируем рефакторинг.

2.5. ПРОЕКТИРОВАНИЕ API С НАСЛЕДОВАНИЕМ ДЛЯ СОКРАЩЕНИЯ ДУБЛИРОВАНИЯ

В этом разделе мы используем метод наследования для сокращения дублирования кода. Самый сложный метод, который мы хотим использовать совместно в обработчиках событий, — `processRequest()`. Обратившись к этому методу в листинге 2.7, заметим, что он использует метод `isTraceEnabled()` для проверки того, должен ли запрос буферизоваться.

Так как между `Trace` и `GraphTrace` существует значительное сходство, общие части можно выделить в новый класс `TraceRequest`, как показывает следующий листинг.

Листинг 2.10. Создание родительского класса `TraceRequest`

```
public abstract class TraceRequest {
    private final boolean isTraceEnabled;

    public TraceRequest(boolean isTraceEnabled) {
        this.isTraceEnabled = isTraceEnabled;
    }

    public boolean isTraceEnabled() {
        return isTraceEnabled;
    }
}
```

← `isTraceEnabled` is shared... — `isTraceEnabled` совместно используется классами `GraphTrace` и `Trace`

С новой структурой оба запроса могут расширять абстрактный класс `TraceRequest`, предоставляя только данные, специфические для каждой разновидности запросов. В следующем листинге показано, как `GraphTrace` и `Trace` могут расширять `TraceRequest`.

Листинг 2.11. Расширение `TraceRequest`

```
public class GraphTrace extends TraceRequest {
    private final int data;

    public GraphTrace(boolean isTraceEnabled, int data) {
        super(isTraceEnabled);
        this.data = data;
    }

    public int getData() {
        return data;
    }
}

public class Trace extends TraceRequest {
    private final String data;

    public Trace(boolean isTraceEnabled, String data) {
```

← `GraphTrace` расширяет `TraceRequest`

← Передает `isTraceEnabled` конструктору родительского класса

← `getData()` для `GraphTrace` возвращает тип `int`

← Класс `Trace` также расширяет `TraceRequest`

```

    super(isTraceEnabled);
    this.data = data;
}

public String getData() {
    return data;
}

```

← `getData()` для `Trace` отличается от экземпляра `GraphTrace`

Рисунок 2.10 показывает, как будет выглядеть иерархия `Trace` и `GraphTrace` после выделения общих частей.

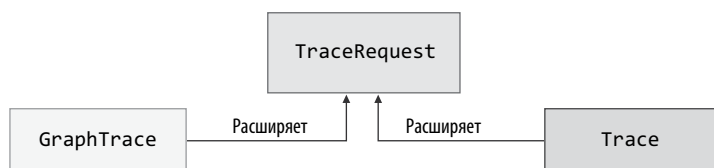


Рис. 2.10. Новый класс `TraceRequest`, который может расширяться классами `GraphTrace` и `Trace` в целях сокращения дублирования кода

Благодаря проведенному рефакторингу можно использовать `TraceRequest` и классы, наследующие от этого класса, в новом базовом классе обработчика, который будет выделен в следующем разделе.

2.5.1. Выделение базового обработчика запросов

Целью рефакторинга было исключение дублирования кода в обработчиках. По этой причине можно выделить новый класс `BaseTraceRequestHandler`, который будет работать с классом `TraceRequest`. Метод `createPayload()`, специфический для типа запроса, размещается в дочерних классах, предоставляющих это конкретное поведение. На рис. 2.11 изображена новая структура.

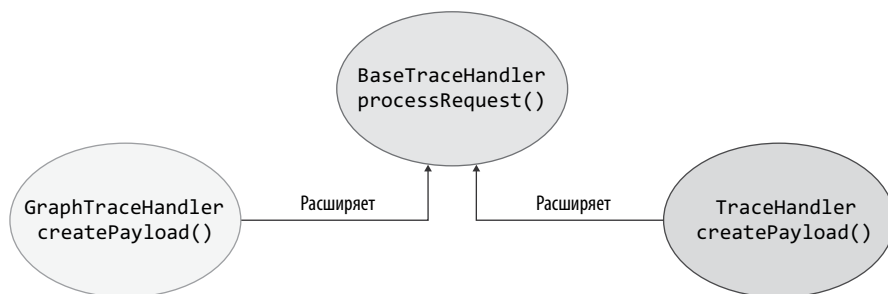


Рис. 2.11. Выделение родительского класса `BaseTraceHandler`

Новый класс `BaseTraceRequestHandler` необходимо параметризовать, чтобы он мог работать с любым классом, расширяющим `TraceRequest`. В следующем листинге приведен переработанный класс `BaseTraceRequestHandler`. Он будет работать с любыми классами, которые вызывают `TraceRequest` или расширяют его. Конструкция `<T extends TraceRequest>` используется в Java для обеспечения этого инварианта.

Листинг 2.12. Создание родительского класса `BaseTraceRequestHandler`

```
public abstract class BaseTraceRequestHandler<T extends TraceRequest> {
    private final int bufferSize;
    private boolean processed = false;
    List<String> buffer = new ArrayList<>();

    public BaseTraceRequestHandler(int bufferSize) {
        this.bufferSize = bufferSize;
    }

    public void processRequest(T trace) {
        if (!processed && !trace.isTraceEnabled()) {
            return;
        }
        if (buffer.size() < bufferSize) {
            buffer.add(createPayload(trace));
        }

        if (buffer.size() == bufferSize) {
            processed = true;
        }

        protected abstract String createPayload(T trace);
        public boolean isProcessed() {
            return processed;
        }
    }
}
```

processRequest получает в аргументе произвольный экземпляр TraceRequest

Он содержит метод isTraceEnabled(), потому что является TraceRequest

Он содержит метод isTraceEnabled(), потому что является TraceRequest

Основная логика обработки такая же, как в решении с дублированием

Теперь логика `processRequest()` работает с любым классом `TraceRequest`. Метод `isTraceEnabled()` будет доступен для него, потому что он определяется классом `TraceRequest`. Обратите внимание, что метод `createPayload()` абстрактный. Конкретная реализация будет предоставляться дочерним классом, который способен обрабатывать запросы `Trace` или `GraphTrace`.

После рефакторинга оба обработчика могут расширить базовый класс, предоставляя только необходимые части кода для своей реализации. Классы `TraceRequestHandler` и `GraphTraceRequestHandler` должны предоставить только реализацию метода `createPayload()`. Родительский класс получает параметр `bufferSize`, который используется основной логикой обработки для ограничения размера буфера. Конструктор дочернего класса должен вызвать конструктор

суперкласса с этим аргументом. Новый класс `TraceRequestHandler` расширяет выделенный базовый класс. Он параметризуется с использованием класса `Trace`, как показано в следующем листинге.

Листинг 2.13. Добавление наследования к `GraphTraceRequestHandler` и `TraceRequestHandler`

```
public class TraceRequestHandler extends BaseTraceRequestHandler<Trace> {

    public TraceRequestHandler(int bufferSize) {
        super(bufferSize);
    }

    @Override
    public String createPayload(Trace trace) {
        return trace.getData() + "-content";
    }
}

public class GraphTraceRequestHandler extends
    BaseTraceRequestHandler<GraphTrace> {

    public GraphTraceRequestHandler(int bufferSize) {
        super(bufferSize);
    }

    @Override
    public String createPayload(GraphTrace graphTrace) {
        return graphTrace.getData() + "-nodeId";
    }
}
```

Предоставляет алгоритм для обработки `GraphTrace`

Применение наследования позволило существенно упростить обработчики. Из них был исключен дублированный код в соответствии с принципом DRY. Обслуживать код стало проще, и при этом усилилась его связанность. После всей проделанной работы можно подумать, что это проектное решение не подразумевает компромиссов. Но это впечатление изменится с поступлением нового бизнес-требования. Эта тема рассматривается в следующем разделе.

2.5.2. Наследование и сильная связанность

Теперь в коде используется наследование, а обработчики предоставляют только метод `createPayload()`. Допустим, поступило новое бизнес-требование: необходимо изменить очередь `GraphTraceRequestHandler`, чтобы он работал с неограниченным значением `bufferSize`. (Хотя в реальных системах создавать неограниченные буферы не рекомендуется, мы рассмотрим этот сценарий для простоты.) Это также означает, что параметр `bufferSize` этому обработчику больше не нужен.

Как известно, логика `processRequest()` находится в родительском классе и совместно используется всеми клиентскими классами. Новое бизнес-требование означает, что метод, ответственный за обработку запросов, можно упростить, как показано в следующем листинге.

Листинг 2.14. Упрощение `processRequest`

```
public void processRequest(T trace) {
    if (!processed && !trace.isTraceEnabled()) {
        return;
    }

    buffer.add(createPayload(trace));
}
```

Логика ограничения количества запросов трассировки в буфере отсутствует

Одна из проблем здесь — метод `processRequest()` можно упростить только для обработчиков трассировки графа. Логика стандартного обработчика должна отслеживать буфер. Таким образом, устранение дублирования вводит в архитектуру сильную связанность. Из-за этого теряется возможность изменить метод `processRequest()` для одного дочернего класса, не влияя на другие дочерние классы. Эта потеря гибкости — вынужденный компромисс, и она существенно ограничивает архитектуру решения.

Одно из возможных решений проблемы — определить особый случай для запросов с использованием `instanceof` без применения буферизации для класса `GraphTrace`. Решение продемонстрировано в следующем листинге.

Листинг 2.15. Использование `instanceof` в обходном решении

```
if(trace instanceof GraphTrace){
    buffer.add(createPayload(trace));
}
```

Такое решение получается слишком хрупким и противоречит изначальной цели введения наследования. Оно создает сильную связанность между родительским и дочерним классами. Внезапно оказывается, что родительский класс должен все знать о типах запросов, которые он должен обрабатывать. Он уже работает не только на уровне обобщенного класса `TraceRequest`. Теперь родительский класс должен знать об одной из фактических реализаций — `GraphTrace`. Логика из конкретного обработчика просачивается в обобщенный обработчик. Таким образом, обработка запросов `GraphTrace` перестает инкапсулироваться в коде, ответственном за обработку запроса.

Проблему можно решить возвратом к дублированию кода. Но в реальной ситуации это вряд ли возможно, потому что перерабатываемые компоненты устроены намного сложнее и требуют приложения гораздо больших усилий.

Вдумчивый читатель увидит, что в рассматриваемом простом примере передача `Integer.MAX_INT` конструктору `GraphTraceRequestHandler` в качестве `bufferSize`

решит проблему. Теоретически это означает, что бизнес-цели создания неограниченного буфера можно достичь без изменения больших фрагментов кода. Однако в реальном мире изменения бизнес-требований, с которыми вы можете столкнуться, будут сложнее. Возможно, вам не удастся осуществить их без сокращения сильной связанности и отказа от наследования.

Я выбрал решение с наследованием из-за контекста, в котором работал исходный код. Допустим, вы хотите разрешить вызывающей стороне передать некоторые части реализации (как в `BaseTraceRequestHandler`). Такая схема называется *паттерном Стратегия (Strategy pattern)*. Возможно, в этом случае проще выбрать наследование, при котором основная логика и заготовки предоставляются в родительском классе, а клиент реализует недостающие части.

Можно использовать другой подход к исключению дублирования — например, применить композицию или паттерн, соответствующий вашим потребностям. Однако у любого решения есть свои плюсы и минусы, и следует проанализировать связанные с ним компромиссы. Необходимо решить, стоит ли желаемая гибкость проблем с обслуживанием дублируемого кода, который может развиваться в разных направлениях. В качестве альтернативы можно рассмотреть применение композиции независимых структурных блоков вместо того, чтобы связывать различные аспекты поведения наследованием.

2.5.3. Компромиссы между наследованием и композицией

Паттерн Стратегия хорошо подходит для нашего примера, если каждый подкласс всегда имеет четко определенный набор требований, частично отделенных друг от друга. Если набор требований растет, возможно, вместо наследования стоит применить композицию. В таком случае требования придется разделить по разным зонам ответственности. В существующей системе уже есть преобразование данных в возможный формат полезной нагрузки и буферизация.

Сейчас буферизация работает относительно прямолинейно и всегда базируется на количестве добавленных в буфер элементов. Предположим, вы хотите применять разные стратегии: бесконечную буферизацию, полное ее отсутствие, существующую схему буферизации с ограничением количества элементов в буфере или, возможно, буферизацию с ограничением объема памяти, учитывающую размер каждого элемента. В решении с наследованием можно воспользоваться методом `tryAddEntry()`; он либо абстрактный, либо имеет реализацию по умолчанию в `BaseTraceHandler`. Будет ли такая схема лучшим выбором?

Разделение функций преобразования данных и буферизации по разным абстракциям (возможно, с повторным использованием функциональных интерфейсов) позволяет коду обработчика быть связующим для абстракций. Это повышает гибкость и дает возможность комбинировать схемы буферизации

с преобразованиями данных. Впрочем, за это приходится расплачиваться увеличением количества абстракций, которые читатель должен понимать при изучении кода. На рис. 2.12 два решения показаны рядом друг с другом для сравнения.

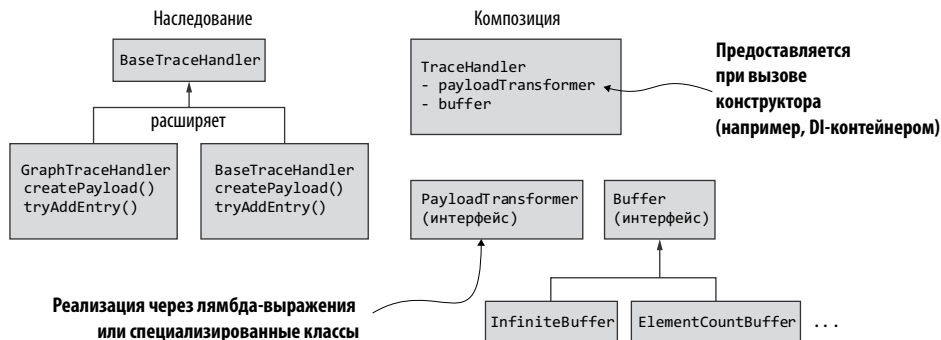


Рис. 2.12. Композиция и наследование в TraceHandler

Если абстракция обработчика в достаточной мере изолируется от остального кода (возможно, благодаря настройке в фазе внедрения зависимостей), а затем просто используется, можно перейти от решения на базе наследования к решению на базе композиции (или наоборот) без вреда для остальной кодовой базы. Все обсуждение того, как избежать дублирования, предполагает, что изначально существует настоящее дублирование. Однако это не всегда так, хотя, на первый взгляд, может создаться именно такое впечатление.

2.5.4. Дублирование внутреннее и ситуативное

В реальном мире программисты склонны подгонять решения под выбранные паттерны. Один из примеров — создание общей абстракции и последующая адаптация кода в нескольких местах для ее распространения. Однако то, что две конструкции выглядят одинаково, еще не значит, что они решают одну задачу. Они также могут по-разному развиваться. Такое дублирование является *ситуативным* — в отличие от *внутреннего* дублирования, изначально присущего коду проекта.

Обычно бывает проще объединить две концепции в одну, если они одинаковые, чем разделить их, если они разные. Если у вас есть абстракция с несколькими применениями, связанность компонентов может быть достаточно высокой. Это означает, что разделить общий код по разным классам может оказаться непросто.

Порой то, что кажется дублированием, на самом деле представляет собой две разные сущности, которые просто одинаково рассматриваются при текущих требованиях, но в дальнейшем могут измениться и не должны считаться эквивалентными. В начале работы над системой бывает сложно различить эти две ситуации.

Иногда подход, при котором мы начинаем с абстракции и адаптации под нее всех возможных применений, не оптимален. Вместо этого можно реализовать систему, создавая независимые компоненты и позволяя им оставаться такими в течение определенного срока (даже если это потребует некоторого дублирования кода). Со временем можно выделять закономерности во взаимодействиях компонентов, и тогда абстракции начнут проявляться. В этот момент уместно исключить дублирование, создав абстракцию (вместо того, чтобы начинать с нее).

В этой главе мы анализировали решения для сокращения дублирования в коде. Мы начали с кода, совместно используемого двумя кодовыми базами, и выделили его в отдельную библиотеку. Были проанализированы компромиссы и проблемы, которые необходимо учитывать на протяжении жизненного цикла библиотеки. Затем был представлен другой подход к совместному использованию кода через специализированный сервис; он может иметь соответствующий API и рассматриваться по принципу «черного ящика». Выделение микросервисов позволило избежать некоторых проблем, присущих решению с библиотекой, но добавило новые сложности и компромиссы. Вторая часть этой главы была посвящена поиску абстракции между двумя компонентами обработчиков, не связанных друг с другом. Мы создали решение на базе наследования, которое предполагает меньше строк кода. Наследование решило некоторые проблемы, но при этом ограничивает гибкость проектирования и подразумевает определенные компромиссы.

Следующая глава посвящена обработке исключений и ошибок в коде. Также вы научитесь обрабатывать исключения из стороннего кода и изучите эффективные приемы обработки исключений в многопоточной среде.

ИТОГИ

- Совместное использование кода в кодовых базах можно реализовать путем выделения библиотеки. И наоборот, повторное использование кода через библиотеку сопряжено с различными проблемами, такими как сильная связанность и ограничение гибкости.
- Выделение общей бизнес-логики в отдельный сервис может оказаться подходящим вариантом для более сложных задач, но приводит к увеличению затрат на обслуживание.
- Наследование помогает исключить дублирование кода и организовать его совместное использование в дочерних классах. К сожалению, наследование сопряжено с многочисленными компромиссами, ограничивающими гибкость кода.
- Иногда есть смысл оставить дублирование кода, потому что это обеспечивает гибкость и сокращает координацию между командами.

Исключения и другие паттерны обработки ошибок в коде

https://t.me/it_boooks

В этой главе:

- ✓ Эффективные паттерны обработки исключений.
- ✓ Исключения из сторонних библиотек.
- ✓ Исключения в многопоточном и асинхронном коде.
- ✓ Исключения в функциональном и объектно-ориентированном программировании.

Ошибки и исключения в коде неизбежны. Почти в каждой программной ветви возможен сбой, если произойдет что-то непредвиденное. Представьте, что вы выполняете простое сложение двух чисел. На первый взгляд ошибок в такой ветви быть не может. Однако следует помнить, что программа выполняет ее в контексте. Например, может возникнуть ошибка нехватки памяти, если на компьютере недостаточно памяти для выполнения какой-нибудь операции. Можно получить исключение прерывания в многопоточном контексте, если код выполняется в отдельном потоке и этот поток получает сигнал прерывания. То есть потенциальные проблемы существуют.

Чаще всего код нетривиален, и в нем возможны самые разнообразные сбои. Обработка исключений — первое, о чем стоит задуматься при создании кода. Код

должен быть *отказоустойчивым*, то есть по возможности восстанавливаться при возникновении ошибок. Прежде чем решать, как обрабатывать исключения, необходимо спроектировать API так, чтобы они выявляли проблемы и явно сигнализировали о них. Тем не менее, если явно выдавать сигнал о каждой возможной ошибке, это затруднит чтение и обслуживание кода.

Не каждый паттерн ошибки требует восстановления работоспособности в коде. Согласно философии «*допущения сбоев*» (*let-it-crash*), впервые определенной в экосистеме Erlang, лучше не восстанавливаться после критических ошибок. В таком сценарии супервизор следит за процессом и в случае сбоя из-за неисправимой ошибки (например, нехватки памяти) просто перезапускает программу. Эта философия не требует применять защитное программирование и пытаться защититься от всех возможных вариантов поведения, приводящего к выдаче исключений. В экосистеме Java такой подход применяется нечасто. Тем не менее некоторые технологии на базе Java — такие, как Akka, — следуют этому паттерну.

В стандартном приложении на базе Java подход допущения сбоев создаст проблемы, потому что обработка запросов пользователей не разделяется на независимые процессы. Типичное Java-приложение содержит n потоков, каждый из которых обрабатывает запросы для некоторых пользователей. Но мы работаем в пределах одного приложения, и если оно полностью будет обрушиваться из-за запроса одного человека, это отразится на других пользователях.

В модели акторов (субъектов), применяемой в Erlang, Akka и других технологиях, обработка имеет более высокую детализацию. Обычно приложение содержит сотни (а то и более) акторов, каждый из которых отвечает за обработку небольшого объема пользовательского трафика. Сбой в одном акторе не повлияет на остальные. Такой подход имеет действительные сценарии применения, но сильно зависит от структуры приложения и его потоковой модели.

Мы рассмотрим плюсы и минусы обоих подходов, а также ситуации, в которых их следует применять. А после разбора самых эффективных паттернов добавим еще один уровень сложности — обработку проблем в асинхронном коде, который работает в многопоточной среде.

Если API проектируется заранее, можно сделать его защищенным и подробным там, где это должно быть выражено явно. Но существуют ошибки, при возникновении которых практически ничего нельзя сделать. Они должны оставаться неявными, и включать их в контракт API не следует. К сожалению, существуют API и сторонние библиотеки, скрывающие проблемы от нас. Мы рассмотрим методы решения таких проблем.

Наконец, мы сравним выдачу исключений в объектно-ориентированной модели с функциональным подходом, использующим монаду Try для решения проблем. Наше путешествие в мир исключений начнется со знакомства с иерархией проблем, о которых может сигнализировать код и которые он может обрабатывать.

3.1. ИЕРАРХИЯ ИСКЛЮЧЕНИЙ

Прежде чем погружаться в более сложные темы, такие как проектирование схемы обработки исключений API, — вкратце рассмотрим иерархию исключений и ошибок, которые часто используются в коде. Эта иерархия изображена на рис. 3.1.

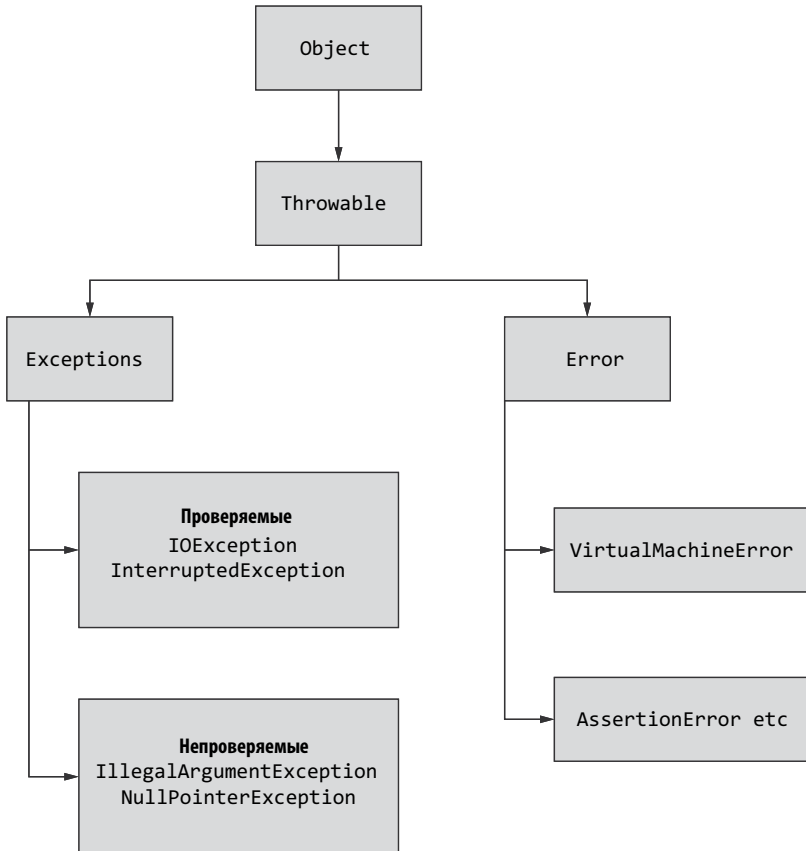


Рис. 3.1. Иерархия исключений в Java

В Java каждое исключение представляет собой объект. Специальный тип `Throwable` (расширяет `Object`) предоставляет полезную информацию о каждой ошибке. Он инкапсулирует причину в сообщении, которое сигнализирует о проблеме. Что еще важнее, он содержит трассировку стека — массив элементов, в котором каждый элемент идентифицирует конкретную строку кода класса, приведшую к исключению. Эта информация необходима для целей диагностики. Она помогает отследить строку, в которой возникла проблема, для устранения ошибки. Далее идут два класса, расширяющих `Throwable` (`Error` и `Exception`). Сообщение `Error` указывает на возникновение критической проблемы, и чаще

всего не следует пытаться перехватывать или обрабатывать его. Например, это может быть ошибка виртуальной машины, которая сигнализирует о критической проблеме с окружением.

В этой главе мы не будем подробно рассматривать обработку ошибок в Java, потому что управлять ею почти невозможно. Впрочем, мы рассмотрим различные стратегии обработки ошибок. Также отмечу, что в оставшейся части главы я буду использовать термины «ошибка» и «исключение» как синонимы для обозначения одной концепции.

В левой части рис. 3.1 изображены исключения, используемые для оповещения о проблемах в коде. Исключение также следует обрабатывать, если есть возможность корректно восстановить нормальную работу программы. Более того, если метод объявляет проверяемое исключение, то компилятор требует, чтобы оно обрабатывалось на стороне вызова (где оно может перехватываться или выдаваться заново). Это означает, что код не будет компилироваться, если исключение не обрабатывается. Например, если при загрузке файлов выдается исключение `IOException`, разумно восстановить работоспособность и попытаться загрузить файл из другого места файловой системы. Позже эти исключения будут использованы для явного проектирования API обработки ошибок.

С другой стороны, обрабатывать непроверяемые исключения не обязательно. Но если код этого не делает, они распространяются в основной поток приложения и приводят к его остановке. Они часто указывают на ошибку использования, после которой восстановиться невозможно, и лучше быстро аварийно прекратить работу, чем пытаться это сделать. Например, если передать отрицательное число в качестве аргумента методу, ожидающему получить положительное число, лучше выдать непроверяемое исключение, потому что попытки восстановления не имеют смысла. Вызывающая сторона также может предпочесть непроверяемые исключения, чтобы, к примеру, упростить их использование из API с функциональным (лямбда) интерфейсом. Это неявная часть кода обработки ошибок.

Концепция проверяемых и непроверяемых исключений также присутствует в других языках, но большинство из них выбирает ту или иную стратегию. Например, в языках программирования Scala и C# каждое исключение рассматривается как непроверяемое; следовательно, перехватывать их не нужно. При этом необходимо действовать осторожно, чтобы не допустить распространения исключений в главный поток приложения. В противном случае программа перестанет работать.

3.1.1. Универсальный и детализированный подход к обработке ошибок

А теперь попробуем понять исключения и их иерархию на эмпирическом уровне. Допустим, имеется метод, который объявляет о выдаче двух проверяемых исключений, как видно из следующего листинга.

Листинг 3.1. Метод, выдающий проверяемые исключения

```
public void methodThatThrowsCheckedException()
    throws FileNotFoundException, InterruptedException
```

Как `FileNotFoundException`, так и `InterruptedException` являются проверяемыми исключениями. Это означает, что сторона вызова этого метода должна обрабатывать их во время компиляции. Первый подход к обработке этих исключений основан на объявлении секции `catch` для обоих типов, как показывает следующий листинг.

Листинг 3.2. Обработка проверяемых исключений

```
public void shouldCatchAtNormalGranularity() {
    try {
        methodThatThrowsCheckedException();
    } catch (FileNotFoundException e) {
        logger.error("File already exists: ", e);
    } catch (InterruptedException e) {
        logger.error("InterruptedException", e);
    }
}
```

Используя два блока `catch`, можно предоставить разное поведение обработки ошибок в зависимости от типа. Часто этот уровень детализации хорошо подходит для обработки исключений.

Из-за того что исключения образуют иерархию, можно изменить блок `catch` для перехвата более широкого типа. Например, `FileNotFoundException` (<http://mng.bz/zQwB>) расширяет `IOException`, так что первый блок `catch` может напрямую перехватывать `IOException`, как видно из следующего листинга.

Листинг 3.3. Обработка проверяемых исключений более широкого типа

```
public void shouldCatchAtHigherGranularity() {
    try {
        methodThatThrowsCheckedException();
    } catch (IOException e) {
        logger.error("Some IO problem: ", e);
    } catch (InterruptedException e) {
        logger.error("InterruptedException", e);
    }
}
```

← `FileNotFoundException`
расширяет `IOException` —
обработать этот тип

В этом листинге есть одна проблема: мы теряем информацию о том, что было выдано именно исключение `FileNotFoundException`. Хотя эти сведения будут доступны на стадии выполнения, во время компиляции можно понять только то, что было выдано исключение `IOException`.

Тип исключения можно расширить до `Exception` или произвольного `Throwable`. Однако при этом появляется риск, что будут перехвачены исключения, которые

изначально перехватывать было не нужно. Обработчик может перехватить критические исключения, не имеющие отношения к процессу, которые следовало бы передать компонентам более высокого уровня.

Если вызываемый метод выдает больше одного исключения, расширяющего `IOException`, можно создать один блок `catch` вместо пары блоков `catch` с меньшей детализацией. Такое решение рационально, если логика обработки ошибок для конкретного типа не требуется и общая обработка нас устраивает.

Если тип исключения не важен, но необходимо перехватывать все проблемы, можно объявить `catch` для всех исключений. Как вы помните из предыдущего раздела, каждое исключение, проверяемое и непроверяемое, расширяет класс `Exception`, так что это решение будет перехватывать все проблемы вызываемого метода. Описанный подход продемонстрирован в следующем листинге.

Листинг 3.4. Перехват всех исключений

```
public void shouldCatchAtCatchAll() {
    try {
        methodThatThrowsCheckedException();
    } catch (Exception e) {
        logger.error("Problem ", e);
    }
}
```

← Перехватываются все исключения, проверяемые и непроверяемые

Преимущество этого решения в том, что можно писать меньше кода, но при этом теряется большое количество информации. Кроме того, следует помнить, что перехватываются будут все исключения — даже те, которые не объявлены как проверяемые исключения, выдаваемые вызванным методом. Возможно, это не то поведение, которого вы ожидали. Вы рискуете перехватить проблему, которая должна распространяться выше по стеку вызовов.

Для сокращения дублирования и сохранения информации об ожидаемых исключениях можно воспользоваться блоком с несколькими `catch`. В следующем листинге в сигнатуре `catch` объявляются исключения `IOException` и `InterruptedException`.

Листинг 3.5. Обработка проверяемых исключений в блоке с несколькими

```
public void shouldCatchUsingMultiCatch() {
    try {
        methodThatThrowsCheckedException();
    } catch (IOException | InterruptedException e) {
        logger.error("Problem ", e);
    }
}
```

В завершение знакомства с исключениями рассмотрим похожий метод, который объявляет два проверяемых исключения, но выдает непроверяемое. Исключение

`RuntimeException` непроверяемое, и оно не должно объявляться в сигнатуре метода, как показывает следующий листинг.

Листинг 3.6. Выдача непроверяемого исключения

```
public void methodThatThrowsUncheckedException()
    throws FileAlreadyExistsException, InterruptedException {
    throw new RuntimeException("Unchecked exception!");
}
```

Листинг 3.4 перехватит эту проблему, даже если вы этого не ожидали. Если сузить блоки `catch`, чтобы они перехватывали только проверяемые исключения, исключение `RuntimeException` не будет перехватываться и будет распространяться. В следующем листинге показано, как исправить положение.

Листинг 3.7. Вызов метода, выдающего непроверяемые исключения

```
public void shouldCatchAtNormalGranularityRuntimeWillBeNotCatch()
    assertThatThrownBy(
        () -> {
            try {
                methodThatThrowsUncheckedException(); ← Выдает непроверяемое
            } catch (FileAlreadyExistsException e) {           исключение
                logger.error("File already exists: ", e);
            } catch (InterruptedException e) {
                logger.error("Interrupted", e);
            }
        })
        .isInstanceOf(RuntimeException.class); ← Исключение RuntimeException
                                                не перехватывается
                                                и распространяется дальше
}
```

Обратите внимание: блоки `catch` перехватывают только исключения, объявленные в сигнатуре `methodThatThrowsUncheckedException()`. Среди них нет блока `catch` для `Exception`, поэтому непроверяемое исключение обработано не будет.

В следующем разделе вы узнаете больше об исключениях и соответствующих им типах Java. Затем мы разберем, как проектировать стратегии обработки исключений для API.

3.2. ЛУЧШИЕ ПАТТЕРНЫ ДЛЯ ОБРАБОТКИ ИСКЛЮЧЕНИЙ В СОБСТВЕННОМ КОДЕ

В подавляющем большинстве случаев код API, который вы пишете для своих продуктов, будет использовать кто-то еще. Если вы работаете в команде, то, скорее всего, разрабатываете логику для одной части системы, а ваши коллеги отвечают за другие ее части.

Точкой интеграции между частями кода должен быть интерфейс, представляющий намерения вашего кода. Неважно, будет ли ваш компонент использоваться только в команде или же вы разрабатываете библиотеку с открытым кодом для широкого круга людей. Проектируя API, следует явно описывать возможные исключения, чтобы вызывающая сторона могла решить, как поступить в случае сбоя. С другой стороны, вы можете разрабатывать компоненты и методы, содержащие внутреннюю логику и не предназначенные для посторонних. Вероятно, в этом случае явно выражать все потенциальные проблемы кода не обязательно.

3.2.1. Обработка проверяемых исключений в общедоступном API

Допустим, вы разрабатываете компонент, предоставляющий общедоступный API, который будут использовать другие члены вашей команды. Когда дело доходит до проверяемых исключений, необходимо явно выразить свое намерение и пометить методы общедоступного API проверяемыми исключениями, которые он может выдать. Например, если вы ожидаете, что в общедоступном методе может произойти сбой из-за проблем с вводом/выводом, объявите исключение в сигнатуре общедоступного API.

Некоторые языки (например, Scala) склонны рассматривать все исключения как непроверяемые, так что методы могут не объявлять их. Если вы проектируете такой API, знайте, что такой подход повышает риск ошибок, потому что клиенты не получают информации о возможных сбоях во время компиляции. Проблема откладывается до времени выполнения, а это значит, что в программе может произойти неожиданный сбой на стадии реальной эксплуатации. Если API объявляет исключения явно, то такая ситуация невозможна, потому что это вынуждает клиента определиться со стратегией обработки исключений на стадии компиляции (то есть при написании кода).

Часто приходится слышать мнение, что объявление пары (двух, трех и даже более) исключений, которые может выдавать API, делает его избыточным, что усложняет написание кода клиентом. Допустим, вы предоставляете доступ к такому методу. Взгляните на следующий листинг.

Листинг 3.8. Метод API с парой исключений

```
void check() throws IOException, InterruptedException;
```

Когда клиент API вызывает этот метод, он должен принять явное решение о его обработке при каждом вызове метода. Если это создает проблемы для вызывающей стороны, то она должна перехватить все исключения с применением паттерна, описанного в предыдущем разделе, и распространить исключение как непроверяемое. Пример дан в следующем листинге.

Листинг 3.9. Распространение исключения как непроверяемого

```

public void wrapIntoUnchecked() {
    try {
        check();
    } catch (RuntimeException e) {
        throw e;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

```

← Перехватывает все исключения при вызове метода
 ← Перехватывает все исключения при вызове метода общедоступного API

В этом листинге `RuntimeException` перехватывается до `Exception`, чтобы избежать лишней упаковки исключения в `RuntimeException`. Также важно упаковать используемое исключение в новое, непроверяемое. При этом вызывающая сторона получает всю информацию о причине возникшего исключения. После этого другие методы в коде могут пользоваться методом, выполняющим упаковку.

Я не рекомендую использовать API, которые скрывают реальные исключения и распространяют непроверяемые исключения в код как решение всех проблем. Такой подход скрывает ожидаемые исключения и делает API менее отказоустойчивым. Тем не менее он показывает, что аргументы против излишней перегруженности API не рациональны. Проверяемые исключения легко преобразуются в непроверяемые.

Если клиенты не хотят обрабатывать ошибки явно, они должны принять осознанное решение игнорировать эти ошибки и распространять их вверх по стеку вызовов. Чаще всего это неверный подход. Здесь становится видно, что объявление проверяемых исключений в сигнатуре метода общедоступных API имеет несколько значительных преимуществ:

- *Такие API явно объявляют свой контракт.* Вызывающая сторона может делать разумные предположения о результате вызова, не обращаясь к реализации метода.
- *Непроверяемые исключения не станут неожиданностью для вызывающей стороны.* Код обработки ошибок писать проще, когда вы точно знаете, какие возможные исключения может выдать вызываемая функция API.

3.2.2. Обработка непроверяемых исключений в общедоступном API

В API часто приходится проверять аргументы и состояние объектов, используемых вызывающей стороной. Если состояние недействительно, можно выдать непроверяемое исключение. Как известно, непроверяемые исключения не обязательно объявлять в сигнатуре метода. Кроме того, их не обязательно обрабатывать в коде на стороне вызова.

Согласно рекомендациям по обработке ошибок (учебное руководство по непроверяемым исключениям в Java доступно по адресу <http://mng.bz/0wXN>), объявление непроверяемого исключения для каждого метода делает код менее понятным. Однако иногда такие исключения допустимо объявлять. Предположим, в API есть метод, который готовит сервис к использованию, как показано в следующем листинге.

Листинг 3.10. Выдача непроверяемого исключения из API

```
boolean running;

public void setupService(int numberOfThreads)
    throws IllegalStateException,
           IllegalArgumentException {
    if (numberOfThreads < 0) {
        throw new IllegalArgumentException(
            "Количество потоков не может быть меньше 0."
        );
    }

    if (running) {
        throw new IllegalStateException(
            "Сервис уже выполняется."
        );
    }
}
```

Объявляет, что метод может выдавать непроверяемые исключения

Если аргумент неверен, выдается исключение IllegalArgumentException

Если сервис уже работает, также может быть выдано исключение IllegalStateException

Исключения, объявленные в сигнатуре метода, предназначены для информации. Вызывающая сторона метода не обязана перехватывать эти исключения, хотя о них полезно знать при взаимодействии с другими API.

Если вы создаете метод, используемый другими компонентами в коде, документируйте предусловия и ожидаемое поведение. К сожалению, разработчики не всегда читают документацию, и она со временем устаревает. Объявление непроверяемых исключений в сигнатуре метода может послужить целям документирования. Вероятность, что разработчик, использующий ваш API, прочитает такую документацию, гораздо выше.

Действительно, объявление слишком большого числа исключений делает код перегруженным и малопонятным. Однако в реальности программные компоненты объявляют только небольшое подмножество методов в виде общедоступного API. Остальные методы, обеспечивающие функционал такого API, скрыты за приватными модификаторами доступа. Эти методы не обязаны быть подробными. Непроверяемые исключения можно удалить из их сигнатур без заметной потери информации.

Чтобы изменять приватные методы конкретного компонента, необходимо знать их внутреннее устройство. Нужно изучить эти методы и знать об

исключениях, которые они могут выдавать. Когда компонент используется по принципу «черного ящика», то есть только через общедоступный API, нельзя требовать, чтобы сторона вызова знала внутреннее устройство компонента. Хорошим решением в таких методах может быть объявление непроверяемых исключений.

Принимая решение о том, должен ли API выдавать проверяемые или непроверяемые исключения, необходимо учитывать множество факторов. Рассмотрим ситуацию, в которой код вызывающей стороны предполагает, что в каждой программной ветви вызываемого API может произойти сбой и выдача исключения. Скорее всего, это означает, что приложение структурировано так, что все исключения перехватываются на более высоком уровне стека вызовов. Можно провести параллель с ситуацией, в которой вы пишете компоненты и API, используемые в коде. В этом случае разумно выдавать непроверяемые исключения. Вам принадлежит как вызывающий код, так и код реализации. Вероятность того, что в вызываемом коде возникнет что-то неожиданное, невелика.

Однако при создании общедоступного API, который может вызываться неизвестным кодом, лучше действовать более явно и объявлять потенциальные проблемы при помощи проверяемых исключений. Такой подход предоставляет потенциальным вызывающим сторонам явную информацию о вашем API. Они будут знать, чего ожидать от вызываемого кода, и смогут защититься от возможных исключений. При явном объявлении исключений в контракте API у вызывающей стороны нет необходимости защищаться от всех потенциальных проблем и пытаться угадать возможные исключения.

Я не могу однозначно сказать, какие типы исключений использовать. Для обоих разновидностей существуют ситуации, в которых они наиболее уместны. Здесь я лишь рассматриваю плюсы и минусы обоих типов. Учтите эти компромиссы и контексте, а затем решите, какие исключения лучше подойдут для вашего кода. В следующем разделе будут рассмотрены некоторые антипаттерны, которые могут снизить отказоустойчивость кода.

3.3. АНТИПАТТЕРНЫ В ОБРАБОТКЕ ИСКЛЮЧЕНИЙ

Допустим, вы создали мощный API, который явно сигнализирует о проблемах и исключениях. Теперь необходимо использовать его и правильно отреагировать на проблемы. К сожалению, в этом сценарии легко потерять информацию или обработать исключения некорректно. Если в API, который вы хотите использовать, объявляются исключения, они должны обрабатываться на стадии компиляции.

Часто возникает искушение проанализировать код и заключить, что такое исключение не может быть выдано ни при каких обстоятельствах. И возможно, на момент анализа это даже будет правдой. Но если метод объявляет непроверяемые

исключения, они должны восприниматься как часть контракта метода. Даже если метод не выдает исключение на время написания кода вызывающей стороны, поведение может измениться в будущем. Следующий листинг демонстрирует этот антипаттерн.

Листинг 3.11. Поглощение исключения

```
try {
    check();
} catch (Exception e) { // Не выдается? Это очень опасно! }
```

Используется метод check() из предыдущего раздела

Вызывающая сторона думает, что исключение не может произойти

Поглощенное исключение не распространяется вверх по стеку вызовов. Кроме того, вы потеряете информацию о нем, что создает риск незаметного сбоя в системе. Отлаживать такие проблемы очень трудно! Никогда не игнорируйте исключение, объявленное в сигнатуре метода. Также может возникнуть искушение ограничиться выводом трассировки стека, как показано в следующем листинге.

Листинг 3.12. Вывод трассировки стека

```
try {
    check();
} catch (Exception e) {
    e.printStackTrace();
}
```

Такой вариант тоже небезопасен, потому что трассировка стека по умолчанию направляет содержимое исключения в стандартный вывод. Однако приемником может быть что-то другое — например, `FileOutputStream`. Кроме того, если стандартный вывод не захватывается и не распространяется, возникает риск потери информации.

Необходимо решить, должно ли исключение обрабатываться на этом конкретном уровне кода. Если да, то при перехвате исключений следует извлекать как можно больше информации. Например, для извлечения информации о `Throwable` можно воспользоваться выводом в журнал, как показано в следующем листинге.

Листинг 3.13. Использование журнала для перехвата ошибки

```
try {
    check();
} catch (Exception e) {
    logger.error("Problem when check ", e);
}
```

Метод `logger.error` извлекает необходимую информацию

Регистратор получает трассировку стека для исключения и распространяет ее на сторону вызова. Трассировка будет присоединена к файлу журнала и поможет вызывающей стороне более эффективно отлаживать проблемы.

Если вы решите обработать определенную ошибку на более высоком уровне, метод, вызывающий `check()`, не должен пытаться перехватывать ее. Вместо этого он должен лишь объявить ее в сигнатуре метода. Явно объявляя исключение в контракте метода, мы сигнализируем клиентам о том, чего им ожидать после вызова метода. Использование этого паттерна позволяет клиентам разработать собственную стратегию обработки ошибок.

3.3.1. Заккрытие ресурсов при возникновении ошибки

Часто код должен взаимодействовать с методами и классами, а для этого нужны системные ресурсы. Например, создание нового файла требует открытия дескриптора файловой системы. Создание клиента HTTP требует открытия сокета, для которого выделяется порт из пула доступных портов. Пока проблем нет и все работает, как ожидалось, необходимо закрыть клиент после завершения процесса.

Рассмотрим простой пример, в котором создается клиент HTTP и выполняются некоторые запросы, после чего клиент закрывается. Код приведен в следующем листинге.

Листинг 3.14. Заккрытие клиента HTTP

```

CloseableHttpClient client = HttpClient.createDefault();
try {
    processRequests(client);
    client.close();
} catch (IOException e) {
    logger.error("Проблема при закрытии клиента или обработке запросов", e);
}

```

На первый взгляд код кажется правильным, и после обработки клиент закрывается. (Здесь обработка включает логику, которая может завершиться неудачей, если в сети будут потеряны некоторые пакеты.) К сожалению, метод `processRequests()` может выдать `IOException`. Если исключение выдается в этой точке кода, то метод `close()` вызван не будет. Возникает риск утечки ресурсов. Это может создать проблемы, если открыть слишком много подключений через сокеты или клиентов.

Необходимо преобразовать этот код, чтобы метод `close()` вызывался даже при сбое в `processRequests()`. Кроме того, проблемы из `processRequests()` необходимо обрабатывать отдельно. Только после их обработки клиент можно закрыть. Следующий листинг показывает, как будет выглядеть такое решение.

Листинг 3.15. Закрытие клиента HTTP при возникновении проблем с обработкой запросов

```

CloseableHttpClient client = HttpClients.createDefault();
try {
    processRequests(client);
} catch (IOException e) {
    logger.error("Проблема при обработке запросов", e);
}
try {
    client.close();
} catch (IOException e) {
    logger.error("Проблема при закрытии клиента", e);
}

```

Перехватывает проблемы
с обработкой запросов

Метод close() вызывается только
после завершения processRequests()

Такой код получается слишком перегруженным и подвержен ошибкам. Первое связано с тем, что нам приходится обрабатывать одно исключение `IOException` дважды. Кроме того, необходимо защититься от сбоя с обработкой и вернуться к вызову метода `close()` даже при возникновении проблем с обработкой. Об этом легко забыть, если API выдает непроверяемое исключение. В таком случае метод `close()` вызываться не будет и возникнет риск утечки ресурсов.

Чтобы немного улучшить этот код, можно воспользоваться конструкцией «try с ресурсами» (try-with-resources), которая возьмет закрытие ресурсов на себя. Такое решение сработает, только если используемый класс реализует интерфейс `AutoCloseable` (см. <http://mng.bz/QWOv>). В листинге 3.16 показано, как автоматически закрыть клиент HTTP с помощью этого механизма.

Листинг 3.16. Закрытие клиента HTTP с использованием «try с ресурсами»

```

try (CloseableHttpClient client = HttpClients.createDefault()) {
    processRequests(client);
} catch (IOException e) {
    logger.error("Проблема при обработке запросов", e);
}

```

Создает HttpClient в конструкции
«try с ресурсами»

Обработывает исключение,
выданное processRequests()

Этот прием позволяет коду на стороне вызова сосредоточиться на логике, которую необходимо выполнить. Управление жизненным циклом объекта, реализующим `Closeable`, осуществляется за нас. Метод `close()` должен предоставить необходимую логику для освобождения ресурсов. Он также выражает намерения кода, позволяя клиентам рассуждать о типах и использовании ими ресурсов.

ПРИМЕЧАНИЕ

Если вы проектируете API так, чтобы ресурсы, выделенные для возврата объекта, освобождались после того, как объект станет не нужен, реализуйте интерфейс `Closeable`.

Хотя абстракция «try с ресурсами» полезна, может оказаться, что она не поддерживается языком программирования. Главная причина для ее

использования — закрытие ресурсов независимо от исхода обработки. Ошибка может помешать дальнейшему выполнению кода, который выдает непроверяемое исключение. В таком случае ресурсы необходимо закрыть после выполнения логики. Из-за этого в некоторых языках предусмотрена возможность выполнения кода независимо от того, выдавалось ли исключение.

В Java для реализации логики, отвечающей за закрытие ресурсов, можно воспользоваться блоком `finally`. Код внутри этого блока выполняется всегда, даже если код выдает исключение. Пример приведен в следующем листинге.

Листинг 3.17. Закрытие ресурсов в блоке `finally`

```
CloseableHttpClient client = HttpClientBuilder.createDefault();
try {
    processRequests(client);
} finally {
    System.out.println("closing");
    client.close();
}
```

Теперь, даже если `processRequests()` выдаст исключение, завершающая логика блока `finally` гарантированно будет выполнена. В этом можно убедиться, так как сообщение о закрытии появится в стандартном выводе.

3.3.2. Антипаттерн использования исключений для управления программной логикой

Другой популярный антипаттерн, встречающийся при реализации объектно-ориентированной обработки исключений, — использование исключений для управления программной логикой (аналог команды `goto`). В таком приложении исключения выходят за рамки нормального применения и выдаются для передачи сигнала вызывающей стороне о том, что логика должна идти по другому пути.

Также возникает соблазн воспользоваться исключением (-ями) для преодоления ограничения «один возвращаемый тип на метод». Допустим, имеется метод, возвращающий `String`. Через какое-то время потребуется изменить его, чтобы он возвращал специальное значение, если строка слишком длинная. На первый взгляд выдача исключения в таком случае кажется правильным решением, и пока это исключение действительно является исключительной ситуацией, оно оправданно. Проблемы начнут проявляться позже, если вызывающая сторона построит условную логику с выполнением разных ветвей программы в зависимости от результата метода.

Чем больше типов исключений выдает метод, тем сложнее становится логика вызывающей стороны. Построение сложной логики на базе исключений обходится дорого (производительность рассматривается в последнем разделе этой

главы). Изменение программных путей в зависимости от исключения усложняет код, затрудняет его понимание и обслуживание.

Допустим, вы хотите, чтобы код заставлял вызывающую сторону обрабатывать граничные случаи в своей логике. Для этого можно применить конструкции функционального программирования, такие как `Try` (она рассматривается далее в этой главе) или `Either`. С их помощью можно спроектировать код обработки граничных случаев, не злоупотребляя исключениями.

Часто при написании кода используются сторонние библиотеки, и невозможно влиять на развитие кодовой базы (или можно влиять крайне ограниченно). В следующем разделе мы обратимся к стратегии обработки ошибок при вызове кода, который нам не принадлежит.

3.4. ИСКЛЮЧЕНИЯ ИЗ СТОРОННИХ БИБЛИОТЕК

При взаимодействии со сторонними библиотеками стратегии обработки исключений следует обдумывать очень тщательно. Рассмотрим пример создания программного компонента, ответственного за сохранение информации о человеке в каталоге.

API будет содержать два общедоступных метода. Первый метод получает информацию о человеке по его имени. Второй — создает информацию об имени человека. Метод `getPersonInfo()` загружает файл из файловой системы, а метод `createPersonInfo()` создает новый файл для данного человека и сохраняет информацию в файле. Клиентский код взаимодействует с API через два общедоступных метода, как показано на рис. 3.2.

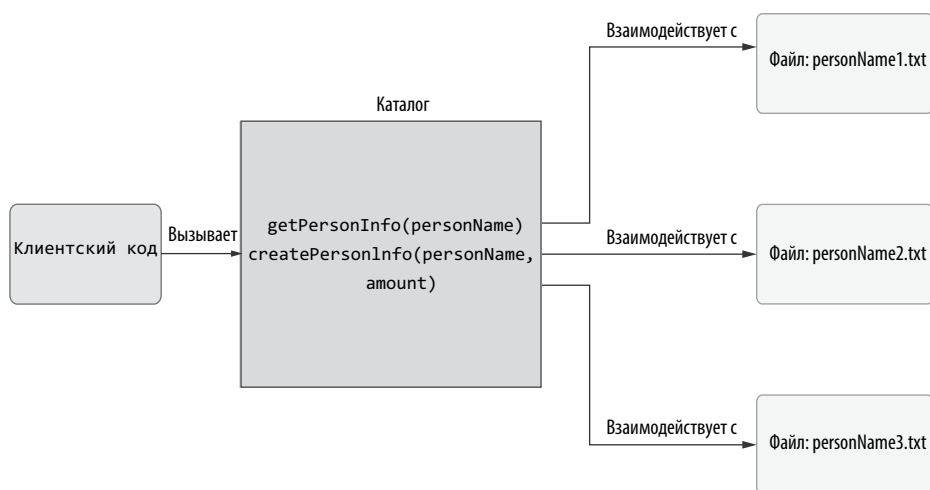


Рис. 3.2. API каталога с информацией о людях, предоставляющий два общедоступных метода

Допустим, вы используете стороннюю библиотеку, которая предоставляет механизм сохранения и загрузки файлов в файловой системе — в данном случае будет использоваться библиотека Apache Commons IO (<http://mng.bz/9KW7>). Библиотека выдает исключения `IOException` или `FileExistsException` (<http://mng.bz/jynr>) при возникновении проблем с любыми операциями, связанными с доступом к файловой системе. Как вы уже знаете, в каждом взаимодействии с файловой системой задействован вызов метода, который может завершиться сбоем. В листинге 3.18 показано, как будет выглядеть API компонента каталогизации.

Листинг 3.18. API с объявленными исключениями

```
import java.io.IOException;
import org.apache.commons.io.FileExistsException;

public interface PersonCatalog {
    PersonInfo getPersonInfo(String personName) throws IOException;
    boolean createPersonInfo(String personName, int amount) throws
        FileExistsException;
}
```

Импортирует сторонний класс, который выглядит подозрительно

Выдает исключение, которое раскрывает информацию о внутренней реализации

Выдает исключение `IOException` из стандартной библиотеки Java

Самое важное, на что следует обратить внимание, — объявления обоих методов API могут выдавать исключения. Метод `getPersonInfo()` выдает исключение `IOException`, доступное в стандартном JDK. Метод `createPersonInfo()` выдает `FileExistsException` — исключение, специфическое для импортированной сторонней библиотеки. И это разумно, потому что оба метода взаимодействуют с файловой системой через стороннюю библиотеку, которая также объявляет эти исключения.

С одной стороны, такое решение работает, как ожидалось: клиенты должны обрабатывать `IOException` и `FileExistsException` в своем коде при взаимодействии с компонентом `PersonCatalog`. С другой стороны, происходит утечка внутреннего исключения, используемого сторонней библиотекой. Распространяя эти исключения и их типы в API, мы создаем сильное связывание между клиентским кодом и сторонней библиотекой, которая используется во внутренней работе компонента `PersonCatalog`. Это противоречит цели введенной абстракции, потому что изменить библиотеку, отвечающую за операции с файловой системой, нельзя. Ее изменение приведет к тому, что в программе могут выдаваться другие исключения, а сигнатура метода API перестанет отражать этот факт.

Также может оказаться, что в другой сторонней библиотеке не будет класса `FileExistsException`, объявленного в сигнатуре метода. С классом `IOException` меньше проблем, потому что он присутствует в JDK, доступном для клиентского кода. Почему бы не удалить команду `throws FileExistsException` и не заменить ее исключением из другой сторонней библиотеки? Так как это открытый интерфейс, изменение данного типа будет означать нарушение совместимости библиотеки. Когда клиенты попытаются использовать новую версию этого метода, код перестанет компилироваться!

Сделаем вывод, что распространение стороннего исключения в открытых методах API кода может оказаться не идеальным решением. Как решить эту проблему? Можно ввести исключение, специфическое для библиотеки, и упаковать используемое исключение в него. Определим класс `PersonCatalogException`, оборачивающий любое исключение, которое выдается сторонней библиотекой, ответственной за взаимодействие с файловой системой. Реализация приведена в следующем листинге.

Листинг 3.19. Создание исключения для конкретной предметной области

```
public class PersonCatalogException extends Exception {
    private PersonCatalogException(String message, Throwable cause) {
        super(message, cause);
    }
    public static PersonCatalogException getPersonException(String personName,
                                                             Throwable t) {
        return new PersonCatalogException("Problem when getting person file for: " +
personName, t);
    }
    public static PersonCatalogException createPersonException(String personName,
                                                                Throwable t) {
        return new PersonCatalogException("Problem when
➤ creating person file for: " + personName, t);
    }
}
```

Приватный конструктор
PersonCatalogException

`PersonCatalogException` получает приватный конструктор, инкапсулирующий реальный объект `Throwable` и сообщение об ошибке. Для метода `getPersonInfo()` имеется фабрика `getPersonException()`, которая создает исключение, специфическое для предметной области. Метод API `createPersonInfo()` представляет похожую ситуацию, в которой нижележащий объект `Throwable` преобразуется в новое исключение `PersonCatalogException`.

После того как специализированное исключение `PersonCatalogException` будет создано, его легко распространить в общедоступном API без раскрытия информации о реальных типах исключений используемой сторонней библиотеки. Такое исключение продемонстрировано в следующем листинге.

Листинг 3.20. PersonCatalog без раскрытия информации о стороннем исключении

```
public interface PersonCatalog {
    PersonInfo getPersonInfo(String personName) throws PersonCatalogException;
    boolean createPersonInfo(String personName, int amount) throws
        PersonCatalogException;
}
```

После таких изменений методы `get` и `create` объявляют, что могут выдавать исключение `PersonCatalogException`. Обратите внимание: стороннее исключение уже не раскрывается, и клиентский код может использовать этот API без сильного связывания с конкретной используемой реализацией. Такое решение обеспечивает высокую гибкость при эволюции API и предоставляет клиентскому коду дополнительную информацию о причине исключения. По одному взгляду на тип исключения вызывающая сторона может определить причину и место, в котором оно было выдано. При использовании низкоуровневых исключений — таких, как `IOException`, — само имя исключения несет куда меньше информации, чем могло бы.

Как видите, упаковка исключений может пригодиться в общедоступном API и в собственном коде, потому что она дает больше контекста возникновения ошибки. По сути, в этом случае передается больше информации с тем же объемом вывода исключений. Это не означает, что необходимо обергивать каждое исключение, распространенное из сторонних библиотек в нашу кодовую базу, но стоит рассмотреть затраты на обслуживание нового исключения и сравнить их с преимуществами, которые они предоставляют. Чаще всего выгода от введения специализированного исключения превышает затраты на его обслуживание. Если вы проектируете приватный компонент, который не будет доступен клиентам напрямую, ничто не мешает использовать исключения без их обергивания.

Заметим, что хотя тип исключения дает вызывающей стороне много информации, сообщение, передаваемое с исключением, должно содержать подробное объяснение случившегося. Кроме того, в исключении сохраняется трассировка стека, и при возникновении аномалий она также предоставляет много информации о том, что пошло не так. Благодаря комбинации этих трех видов информации (тип исключения, сообщение и трассировка стека) проще определить причину ошибки. Тип исключения также полезен для компилятора. Когда ошибка происходит во время выполнения, желательно также располагать всей этой информацией.

До этого момента мы занимались проектированием синхронного кода. В следующем разделе речь пойдет о коде, работающем в многопоточной и асинхронной модели.

3.5. ИСКЛЮЧЕНИЯ В МНОГОПОТОЧНЫХ СРЕДАХ

Обработка исключений в многопоточном и однопоточном контекстах происходит по-разному. В первом случае при отправке нового действия исполнителю необходима обратная связь об успехе или неудаче. Без механизма получения этой информации возникает риск того, что асинхронное действие, выполняемое в отдельном потоке, завершится сбоем без сигналов об этом. Такие незаметные сбои опасны, и их трудно диагностировать.

При взаимодействии с исполнителем возможны два способа отправки работы. Можно запланировать новое выполняемое действие при помощи метода `submit()`, который возвращает экземпляр `Future`, а затем воспользоваться этим экземпляром `Future` для получения результата действия. Во втором варианте планирования асинхронных операций используется метод `execute()`. По сути, этот метод работает по принципу «выстрелил и забыл», то есть результат из таких действий мы не получаем.

Получение результата действия означает, что он либо успешен, либо неудачен, если в коде выдано исключение. В следующем листинге показано, как работает код обработки исключений при отправке действия.

Листинг 3.21. Отправка с ожиданием

```

ExecutorService executorService = Executors.newSingleThreadExecutor();
Runnable r =
    () -> {
        throw new RuntimeException("problem");
    };
Future<?> submit = executorService.submit(r);
assertThatThrownBy(submit::get)
    .hasRootCauseExactlyInstanceOf(RuntimeException.class)
    .hasMessageContaining("problem");

```

Используется исполнитель с одним отдельным рабочим потоком

Отправленное действие выдает непроверяемое исключение

submit() возвращает Future

get() подразумевает блокировку главного потока

Важно отметить, что метод `get()` блокирует выполнение и блокирующая операция должна завершиться при выполнении `get()`. Если используемое действие завершается с исключением, оно будет распространено в главный поток. Если действие отправлено, но результат в коде не используется («выстрелил и забыл»), то исключение не распространяется и возникает риск незаметного сбоя. Следует помнить, что, если сервис-исполнитель возвращает `Future`, необходимо проверить его правильность.

ПРИМЕЧАНИЕ

Информацию об интерфейсе `Future` можно найти по адресу <http://mng.bz/W70a>. Для разработчиков .NET интерфейс `Future` аналогичен `Task`.

Способ исполнения несколько отличается, потому что он не возвращает никакого результата. Его реализация представлена в следующем листинге.

Листинг 3.22. «Исполнил и забыл»

```

Runnable r =
    () -> {
        throw new RuntimeException("problem");
    };
executorService.execute(r)

```

Если исполнитель не возвращает результат, есть вероятность, что сбой асинхронного действия, выполненного в отдельном потоке, пройдет незаметно. Такое исключение может привести к остановке работы потока. Это создаст проблему, если используется пул потоков с фиксированным их количеством. При сбое в потоке может оказаться, что он не будет воссоздан. Кроме того, возникает риск, что в какой-то момент во всех потоках произойдут сбои и пул опустеет. Если пул потоков адаптируется к трафику, возникает риск утечки ресурсов. Каждый поток занимает значительный объем памяти, и создание большого количества новых потоков может привести к ее нехватке.

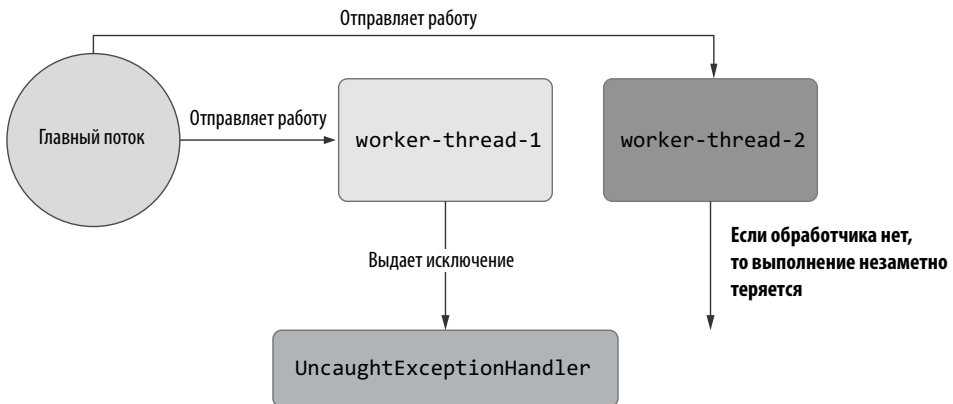


Рис. 3.3. Глобальная обработка исключений в многопоточном контексте

Как показано на рис. 3.3, главный поток отправляет работу потоку `worker-thread-1` (рабочий поток-1) с использованием метода `execute()`, не получая обратно объект обещания (`promise`). Затем рабочий поток выполняет это действие асинхронно. Если обещание не возвращается, главный поток не может сообщить нам о проблемах, возникших при обработке `worker-thread`. К счастью, можно зарегистрировать глобальный обработчик исключений, который активизируется при возникновении любых исключений во время обработки. Если обработчика нет (как в `worker-thread-2` (рабочий поток-2)), появляется риск, что исключение будет незаметно потеряно, а рабочий поток может остановиться, что приведет к описанной ранее утечке ресурсов.

Рассмотрим модульный тест, который проверяет логику глобального обработчика исключений. Вызовем метод `execute()` с действием, в котором произойдет сбой. Затем тест проверит, что обработчик `UncaughtExceptionHandler` был вызван при возникновении исключения. В следующем листинге показан этот сценарий использования.

Листинг 3.23. Регистрация `UncaughtExceptionHandler`

```
// Дано
AtomicBoolean uncaughtExceptionHandlerCalled = new AtomicBoolean();
ThreadFactory factory =
    r -> {
        final Thread thread = new Thread(r);
        thread.setUncaughtExceptionHandler(
            (t, e) -> {
                uncaughtExceptionHandlerCalled.set(true);
                logger.error("Exception in thread: " + t, e);
            });
        return thread;
    };

Runnable task =
    () -> {
        throw new RuntimeException("problem");
    };
ExecutorService pool = Executors.newSingleThreadExecutor(factory);
// Когда
pool.execute(task);
await().atMost(5,
    TimeUnit.SECONDS).until(uncaughtExceptionHandlerCalled::get);
```

Присваивается true, если очередь выполняется

Назначает глобальный обработчик исключений

Возникло исключение, поэтому `uncaughtExceptionHandlerCalled` присваивается true

Назначает время ожидания до вызова обработчика (все вызовы асинхронны)

`execute()` активизирует действие в отдельном рабочем потоке

Как видите, обработка исключений в многопоточной среде — нетривиальная задача, особенно если используемый API разрешает или вынуждает асинхронно инициировать действие и забыть о результатах. Но если можно получить объект `Future`, обертывающий результат выполнения, следует использовать этот API, потому что он вынуждает учитывать результат и возможность исключения.

API обещаний — хорошо известная конструкция, позволяющая создавать асинхронный код и динамично формировать асинхронные операции. В этом Java API присутствует конструкция `CompletableFuture` для создания динамичных асинхронных API, явно сохраняющих информацию о сбоях (см. <http://mng.bz/NxMn>). Аналогичные API также встречаются в других языках программирования. Посмотрим, как обрабатывать исключения с использованием API обещаний Java.

3.5.1. Исключения в асинхронной программной логике с API обещаний

В идеале при создании асинхронного рабочего процесса вы будете взаимодействовать с вводом/выводом, сетями и другими внешними ресурсами с использованием API, который работает по асинхронной схеме. Такой API должен возвращать обещание, которое может использоваться для формирования цепочек асинхронных операций. К сожалению, в реальном мире иногда приходится создавать преобразующую прослойку между синхронными и асинхронными API.

Допустим, необходимо вызвать какой-то внешний сервис. Метод, ответственный за вызов внешнего сервиса, работает синхронно, так что его придется упаковать в `CompletableFuture` API, позволяющий изменить последующую асинхронную схему. Внешний вызов включает операцию ввода/вывода, поэтому он объявляет о возможной выдаче `IOException`.

Так как исключение `IOException` является проверяемым, необходимо обработать его асинхронным образом. Воспользуемся методом `supplyAsync()` — он упаковывает блокирующий вызов и возвращает неблокирующий тип. Этот тип распространяется к вызывающим сторонам, которые ожидают асинхронной операции. Первое возможное решение — упаковка проверяемого исключения в непроверяемое для распространения его к вызывающей стороне, как показывает следующий листинг.

Листинг 3.24. Упаковка исключения в асинхронный API

```
public int externalCall() throws IOException {
    throw new IOException("Проблема при
    вызове внешнего сервиса");
}

public CompletableFuture<Integer> asyncExternalCall() {
    return CompletableFuture.supplyAsync(
        () -> {
            try {
                return externalCall();
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        });
}
```

Методы `externalCall()` могут выдавать `IOException`

Выдает новое исключение `IOException` для моделирования сбоя

Упаковывает синхронный вызов и возвращает `CompletableFuture`

Упаковывает исключение `IOException` в непроверяемое

Важно отметить, что мы распространяем `IOException` из используемой библиотеки напрямую, без создания специализированного исключения-обертки. Это сделано для простоты примера. Плюсы и минусы такого решения описаны подробно в разделе 3.4.

Подход с упаковкой и распространением непроверяемого исключения далеко не идеален. Мы смешиваем две абстракции: первая — API обещаний, инкапсулирующий результат, который может быть исполнен в будущем, или исключение, если действие завершится сбоем. Другая абстракция синхронно выдает исключение, которое будет распространяться на вызывающую сторону. Java API упаковывает его в исключение `CompletionException` — оно захватывается в пуле потоков, где выполняется асинхронное действие.

При вызове метода `asyncExternalCall()` вы увидите трассировку стека, которая сообщает, что исключение было распространено на несколько уровней API `concurrent`. Только после этого вы сможете понять суть проблемы. В следующем листинге приведена трассировка стека.

Листинг 3.25. Трассировка стека для исключения, которое не было правильно обработано

```

java.util.concurrent.CompletionException: java.lang.RuntimeException:
java.io.IOException: Problem when calling an external service ←
    at java.util.concurrent.CompletableFuture.encodeThrowable(
CompletableFuture.java:273)
    at java.util.concurrent.CompletableFuture.completeThrowable(
CompletableFuture.java:280)
    at java.util.concurrent.CompletableFuture$AsyncSupply.run(
CompletableFuture.java:1592)
    at java.util.concurrent.CompletableFuture$AsyncSupply.exec(
CompletableFuture.java:1582)
    at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:289)
    at java.util.concurrent.ForkJoinPool$WorkQueue.runTask(
ForkJoinPool.java:1056)
    at java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1692)
    at java.util.concurrent.ForkJoinWorkerThread.run(
ForkJoinWorkerThread.java:157)
Caused by: java.lang.RuntimeException: java.io.IOException:
➔ Проблема при вызове внешнего сервиса ←

```

Вызовы библиотечных функций, которые должны обрабатывать неожиданные ситуации

После кода библиотеки обнаруживается причина ошибки

Такая трассировка стека указывает, что сбои были обработаны неправильно, потому что она включает много промежуточных шагов для обработки ситуации в библиотеке параллельного выполнения. В зависимости от языка или библиотеки такое исключение либо не распространяется, либо уничтожает поток, приводя к утечке ресурсов. Как обработать ошибки и совместить их с API обещаний?

Для решения этой проблемы можно создать новый экземпляр `CompletableFuture`, возвращающий результат или исключение. Здесь критичен второй случай. В следующем листинге обещание заполняется исключением, но оно не выдается.

Листинг 3.26. Исполнение обещания с результатом или исключением

```

CompletableFuture<Integer> result = new CompletableFuture<>();
CompletableFuture.runAsync(
    () -> {
        try {
            result.complete(externalCall());
        } catch (IOException e) {
            result.completeExceptionally(e);
        }
    });
return result;

```

Внешний вызов завершился успешно и исполнил обещание

Новое обещание `CompletableFuture`, которое еще не исполнено

Если происходит исключение, то оно упаковывается в обещание

Возврат результата, который будет исполнен значением или исключением

Вызывающая сторона приведенного выше метода получает значение или перво-причину ошибки, упаковывая исключение в API обещаний. Вы не увидите трассировки стека, относящейся к библиотеке параллельного выполнения, поскольку мы не выдавали исключение заново. Благодаря этому отсутствует риск того, что это исключение уничтожит поток или останется незамеченным.

ПРИМЕЧАНИЕ

Метод, представленный в этом разделе, типичен для многих асинхронных API; он будет полезен и в выбранном языке.

В следующем разделе мы сравним наш объектно-ориентированный механизм обработки ошибок с использованием исключений и подход, применяемый в функциональном программировании. Также будет рассмотрена конструкция Try, инкапсулирующая успех или сбой при выполнении; она напоминает только что рассмотренный API обещаний.

3.6. ФУНКЦИОНАЛЬНЫЙ ПОДХОД К ОБРАБОТКЕ ОШИБОК С TRY

До сих пор мы рассматривали объектно-ориентированный подход к обработке ошибок. Теперь рассмотрим функциональный подход к управлению ошибками. Сосредоточимся на одном из главных аспектов функционального программирования: коде, свободном от побочных эффектов.

Когда метод выдает исключение, это значит, что у него есть побочные эффекты. Если у вас есть простой метод, который возвращает значение и выдает исключение, можно сделать вывод, что исключение выдается в объявлении метода. Этот паттерн использовался в объектно-ориентированном программировании, но следует помнить, что исключение — это побочный эффект. Вызывающая сторона должна обработать фактически возвращенное значение, но она также должна защититься от исключений. Когда исключение явно упоминается в контракте метода, функциональный код знает, чего следует ожидать, и может защититься от этого побочного эффекта, упаковав его в Try (вскоре мы рассмотрим этот вопрос подробнее).

С другой стороны, когда выдаваемое исключение является непроверяемым и не объявляется в контракте метода, вызывающая сторона может не обработать его и побочный эффект будет распространяться вверх по стеку вызовов. Отсутствие обработки может быть обусловлено тем, что вызывающая сторона не ожидает исключения и, как следствие, не защищается от него. В функциональном программировании такое поведение создает проблемы.

Главный принцип функционального программирования заключается в моделировании всех возможных результатов вызова функции в типе. Если вызываемая функция может создать сбой, этот результат должен моделироваться возвращаемым типом функции и объявленным исключением. Выдача исключения описывается явно при использовании проверяемого исключения, но может быть и неявной, если метод выдает непроверяемое исключение. Такое непоследовательное поведение в функциональном программировании недопустимо. Тип,

возвращаемый функцией, должен моделировать все ее возможные результаты. Именно по этой причине монада `Try` (также называемая монадой `Error`) критична при моделировании обработки ошибок в функциональном программировании (см. <http://mng.bz/la42> и <http://mng.bz/BxV1>).

Рассмотрим простую конструкцию. `Try` может передавать одно из двух возможных состояний: успех или сбой. Тип может находиться в одном или другом состоянии, но никогда в обоих сразу. Возможные состояния показаны на рис. 3.4.

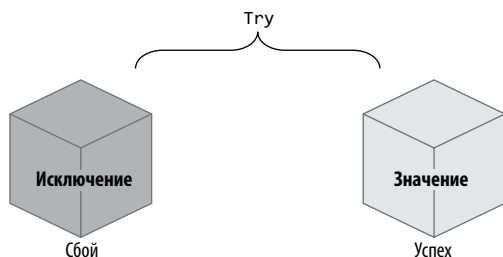


Рис. 3.4. Монада `Try`

В предыдущем разделе был показан тип обещания, использующий API `CompletableFuture`. Этот тип похож на него, потому что он передает результат асинхронного вычисления или возвращает признак сбоя, показывая, что происходит во время обработки API. У него есть одно существенное ограничение — он должен использоваться только в контексте асинхронного программирования.

Однако монада `Try` может инкапсулировать состояние обработки как в синхронном, так и в асинхронном контексте. Тип `Try` более универсален и гибок, благодаря чему он служит основной абстракцией для представления успеха или сбоя в функциональном программировании. Посмотрим, как функциональный подход к обработке ошибок будет выглядеть на языке программирования Java. Воспользуемся библиотекой `Vavr` (<https://www.vavr.io/>), которая предоставляет доступ к типу `Try`.

ПРИМЕЧАНИЕ

Если ваш метод возвращает тип `Try`, вызывающая сторона этого метода всегда должна быть готова к тому, что в методе может произойти сбой. Дальнейшая обработка может объединяться в цепочки с методами функционального программирования — `filter` или подобными.

Посмотрим, как вызывающая сторона метода, возвращающего `Try`, реализует обработку. Действие клиента представляет задачу, в которой может произойти сбой; следовательно, он может выдать исключение. Мы смоделируем его для целей этого теста, чтобы продемонстрировать поведение `Try`, когда упакованный

вызов выполняется без сбоя, как показано в следующем листинге. В реальной системе вызовы компонентов или внешних систем, в которых может произойти сбой, будут упаковываться.

Листинг 3.27. Монада Try для успешного выполнения

```
// Дано
String defaultResult = "default";
Supplier<Integer> clientAction = () -> 100;

// Если
Try<Integer> response = Try.ofSupplier(clientAction);
String result = response.map(Object::toString).getOrElse(defaultResult);

// То
assertTrue(response.isSuccess());
response.onSuccess(r -> assertThat(r).isEqualTo(100));
assertThat(result).isEqualTo("100");
```

Следует отметить, что точка интеграции с действием, в котором может произойти сбой (клиентское действие), упаковывается в тип `Try`. Абстракция `Try` должна возвращаться методами, в которых возможен сбой. Вызывающая сторона может присоединить последующую обработку к типу `Try` вместо того, чтобы беспокоиться об исключениях. Монада `Try` инкапсулирует исключение, если оно произойдет.

Если вы хотите извлечь фактическое значение `String` из `Try`, для его получения из монады можно воспользоваться методом `getOrElse()`. Однако если монада `Try` содержит исключение, она не имеет возвращаемого значения. В такой ситуации необходимо предоставить результат по умолчанию. Он будет возвращаться, если в упакованном в `Try` действии происходит сбой. Эта задача решается вызовом метода `getOrElse()`.

Если нужно создать процесс в зависимости от того, было ли действие успешно выполнено или нет, для проверки можно воспользоваться методом `isSuccess()` (потому что абстракция `Try` является конструкцией функционального программирования). Функциональную обработку можно объединять в цепочки с использованием таких методов, как `map`. Если `Try` сообщает об успехе, вызывается `map`. В противном случае `map` не вызывается. При успехе обратный вызов выполняется, только если он содержит значение, а не ошибку.

Одно из главных преимуществ использования монады `Try` заключается в том, что исключения в стандартных блоках `try-catch` обрабатывать не приходится. Они все еще перехватываются, но только средствами API `Try` из функционального программирования. При этом код обработки исключений не загромождает бизнес-логику. Посмотрите, как работает функциональная обработка ошибок, если упакованное действие выдаст исключение.

Когда в клиентском действии возникает сбой, та же самая абстракция `Try` взаимодействует с системой типов. Обратите внимание, что на этот раз `clientAction` выдает исключение. Таким образом, мы имитируем вызов отказавшего компонента. Это можно использовать, чтобы проверить, как сейчас выглядит абстракция `Try`.

В листинге 3.28 приведен тот же тип `Try` с упаковкой действия. На этой стадии никаких различий в обработке нет. Сторона, работающая с API, должна взаимодействовать с компонентом, в котором может произойти сбой, только через тип `Try`, если хочет создать логику, не загроможденную многочисленными блоками `try-catch`.

Листинг 3.28. Монада `Try` для сбоя

```
Supplier<Integer> clientAction =
    () -> {
        throw new RuntimeException("problem");
    };

// Когда
Try<Integer> response = Try.ofSupplier(clientAction);
String result = response.map(Object::toString).getOrElse(defaultResult);
Option<Integer> optionalResponse = response.toOption();

// Тогда
assertTrue(optionalResponse.isEmpty());
assertTrue(response.isFailure());
assertThat(result).isEqualTo(defaultResult);

response.onSuccess(r -> System.out.println(r));
response.onFailure(ex -> assertTrue(ex instanceof RuntimeException));
```

Обратите внимание: функциональная обработка, которую мы хотим выполнить, объединяется в цепочки так же, как прежде. Наша логика использует метод `map()` для выполнения некоторого действия в том случае, если действие клиента завершилось успехом. Однако на этот раз `map()` не вызывается, потому что действие клиента выдало исключение. В нашем случае `Try` содержит признак сбоя. Следовательно, при вызове `getOrElse()` будет возвращено значение по умолчанию. Метод не может вернуть обработанное значение, потому что его нет.

`Try` можно преобразовать в `Option` (конструкция библиотеки `Vavr`, сходная с `Optional` в Java) — еще один тип функционального программирования. Он сигнализирует о присутствии или отсутствии значения. В целом `Option` напоминает `Try`, но не содержит причины, по которой значение может оказаться пустым. Некоторые функциональные API работают с типом `Option`. Преобразование позволяет легко организовать интеграцию между этими API. Вызывающая сторона `Try` может проверить наличие сбоя методом `isFailure()` по аналогии с предыдущим примером, в котором выполнялась проверка `isSuccess()`. Здесь можно использовать обе проверки. Наконец, два функциональных процесса объединяются в цепочку.

При использовании Try в реальном коде вызывающая сторона должна обрабатывать как успехи, так и сбои. Например, это можно сделать созданием методов `onSuccess()` и `onFailure()`. В нашем случае при моделировании сбоя будет выполнен только второй обратный вызов, который извлекает информацию об истинной причине проблемы.

3.6.1. Использование Try в рабочем коде

Рассмотрим пример использования Try, приближенный к действительности. Допустим, необходимо выполнить запрос HTTP к внешнему сервису. Этот сервис возвращает данные в формате JSON. Нужно извлечь из JSON только идентификатор. Для этого необходимо выполнить несколько операций, в ходе которых могут произойти сбои с выдачей исключений.

Первое действие — внешний вызов HTTP. После него необходимо извлечь строковый контент из тела сущности HTTP. Извлечение может завершиться с ошибкой, потому что в нем задействованы операции ввода/вывода. Наконец, необходимо отобразить строковый контент на класс сущности Java. Эта операция тоже может завершиться неудачей, потому что контент строки десериализуется в JSON. Имея сущность, можно извлечь идентификатор.

Такую обработку легко объединить в цепочку с использованием API Try. Сначала нужно упаковать клиентский вызов в монаду Try, инкапсулирующую результат обработки. Затем каждую стадию обработки можно выразить с использованием API Try. Если желаемое действие выдает непроверяемое исключение, это действие следует выполнить в методе `mapTry()`. При выдаче исключения тип Try исполняется, а вся ветвь обработки помечается как сбойная. Результат показан в следующем листинге.

Листинг 3.29. Вызов сервиса HTTP с использованием Try

```
private static final Logger logger =
    LoggerFactory.getLogger(HttpCallTry.class);

public String getId() {
    CloseableHttpClient client = HttpClients.createDefault();
    HttpGet httpGet = new HttpGet("http://external-service/resource");
    Try<HttpResponse> response = Try.of(() -> client.execute(httpGet));
    return response
        .mapTry(this::extractStringBody)
        .mapTry(this::toEntity)
        .map(this::extractUserId)
        .onFailure(ex -> logger.error("The getId() failed.", ex))
        .getOrElse("DEFAULT_ID");
}

private String extractUserId(EntityObject entityObject) {
    return entityObject.id;
}
```

Внешний вызов упаковывается в монаду Try

Используем mapTry(), потому что extractStringBody() выдает исключение

На последней стадии обработки извлекается идентификатор

Регистрирует исключение при возникновении проблем

Возвращает значение по умолчанию, если на любой стадии обработки произошел сбой

```
private String extractStringBody(HttpResponse r) throws IOException {
    return new BufferedReader(
        new InputStreamReader(r.getEntity().getContent(),
            StandardCharsets.UTF_8))
        .lines()
        .collect(Collectors.joining("\n"));
}

private EntityObject toEntity(String content) throws JsonProcessingException {
    return OBJECT_MAPPER.readValue(content, EntityObject.class);
}

static class EntityObject {
    String id;

    public EntityObject(String id) {
        this.id = id;
    }
}
```

Только просмотр определения обработки позволяет сделать вывод, на каких стадиях есть вероятность сбоя, а на каких нет. Вызовы `extractStringBody()` и `toEntity()` могут завершиться сбоем. Взглянув на объявление метода `extractStringBody()`, вы заметите, что он объявляет исключение `IOException`, которое должно быть обработано вызывающей стороной. Аналогичным образом метод `toEntity()` может выдать исключение `JsonProcessingException`. Когда все действия, которые могут завершиться сбоем, совершены, извлекается идентификатор пользователя. Наконец, метод `getId()` должен вернуть тип `String`. В этой ситуации вызывающая сторона метода ничего не знает о монаде `Try`, которая используется во внутренней обработке.

Когда потребуется извлечь строку из монады `Try`, возможны два варианта. Здесь используется метод `getOrElse()`. Если обработка завершается успехом, она просто возвращает правильный идентификатор пользователя. Но если на одной из стадий обработки произойдет сбой, можно предоставить вызывающей стороне разумное значение по умолчанию. Исключение (если оно происходит) регистрируется в журнале методом `onFailure()`. Если предоставить разумное значение по умолчанию нельзя, можно попробовать вернуть тип `Try` из `getId()`, чтобы вызывающая сторона сама обработала его; вероятно, это лучшее решение.

Наконец, функциональную обработку, основанную на `Try`, можно преобразовать в стандартный паттерн выдачи исключений `getOrElseThrow()`. Если монада `Try` содержит исключение, оно выдается на сторону вызова. У последнего решения есть несколько недостатков, которые будут рассмотрены в следующем разделе. Тем не менее перед этим сравним подход с `Try` со стандартной реализацией Java на базе исключений, как показано в следующем листинге.

Листинг 3.30. Вызов сервиса HTTP с использованием API Exception

```

public String getIdExceptions() {
    CloseableHttpClient client = HttpClients.createDefault();
    HttpGet httpGet = new HttpGet("http://    /external-service/resource");
    try {
        CloseableHttpResponse response = client.execute(httpGet);
        String body = extractStringBody(response);
        EntityObject entityObject = toEntity(body);
        return extractUserId(entityObject);
    } catch (IOException ex) {
        logger.error("The getId() failed", ex);
        return "DEFAULT_ID";
    }
}

```

Используемая логика похожа на решение с Try. Единственное отличие в том, что приходится создавать промежуточные переменные, используемые на следующем шаге обработки. Решение с Try более функционально, и в нем можно передавать ссылки на функции (лямбда-выражения).

Главное отличие стандартного подхода try-catch от функционального подхода — возвращаемый тип метода. С функциональным подходом можно вернуть Try<String> и предоставить вызывающей стороне решать, что делать со сбоем. Возможность сбоя передается компилируемым типом (Try) и должна быть обработана; в противном случае код не будет компилироваться. Логика на базе исключений менее явно выражена и не позволяет вернуть один тип, инкапсулирующий успех или сбой операции. Вызывающая сторона должна обработать результат String и защититься от возможного исключения. Существуют разные принципы обработки исключительных ситуаций. Рассмотрим наиболее частые ловушки, с которыми сталкиваются программисты при внедрении абстракции Try с API, использующими Exception.

3.6.2. Объединение Try с кодом, выдающим исключение

Главное, на что следует обратить внимание, — вызывающая сторона должна взаимодействовать с компонентом, в котором могут произойти сбои, через Try. Использование абстракции Try позволяет смоделировать каждый возможный исход (успех или сбой) в системе типов. К сожалению, при введении функционального программирования для обработки ошибок в языках, использующих исключения в качестве основного механизма уведомления о сбоях, возникают проблемы. При выборе механизма обработки исключений — функционального программирования с Try или объектно-ориентированного с исключениями — следует придерживаться одного варианта и последовательно применять его в кодовой базе. Объединение двух решений затрудняет анализ кода. Необходимо обрабатывать оба состояния Try (успех или сбой), но также придется использовать паттерны try-catch для перехвата исключений.

Как вы помните, непроверяемые исключения могут выдаваться любым методом. Объявлять их в сигнатурах методов не нужно. Из-за этого упаковка в `Try` каждого возможного метода, который может завершиться сбоем, становится проблематичной. Представьте, что у вас имеется логика, которая взаимодействует с другими компонентами, и любой из них может выдать непроверяемое исключение. В этом сценарии каждый вызов к каждому компоненту должен быть упакован в тип `Try`. Такой код плохо читается и получается слишком длинным.

Когда мы вызываем функциональный код из нефункционального, мы должны преобразовывать его в паттерн `try-catch`. При вызове нефункционального кода из функционального необходимо перехватывать все возможные исключения и инкапсулировать их в монады `Try` для устранения побочных эффектов.

В разделе, посвященном проектированию открытых API, я упоминал о том, что часто бывает полезно объявлять все исключения (проверяемые и непроверяемые) в сигнатурах методов. Если вы взаимодействуете с подобным компонентом, проще упаковать такой API в функциональную конструкцию `Try`. Все объявляется явно, и, если выбрать функциональный подход к обработке ошибок, вам будет проще упаковывать только те методы, которые выдают исключения. С другой стороны, предположим, что подход функционального программирования интегрируется с API, выдающим непроверяемые исключения, не объявленные в сигнатуре метода. В итоге почти каждый вызов придется упаковывать в монаду `Try`, из-за чего синтаксис кода станет слишком перегруженным и плохо читаемым.

Можно заключить, что функциональный подход к обработке ошибок лучше всего работает при использовании системы с явной типизацией. Если такой подход соответствует вашему стилю программирования, применение `Try` принесет пользу. К сожалению, создать унифицированную систему обработки исключений непросто, если вызываемые API злоупотребляют непроверяемыми исключениями. В следующем разделе сравнивается производительность разных стратегий обработки исключений.

3.7. СРАВНЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ КОДА ОБРАБОТКИ ИСКЛЮЧЕНИЙ

Наконец, сравним стратегии обработки исключений с точки зрения производительности. Воспользуемся системой для создания микробенчмарков JMH (Java Microbenchmark Harness), которая позволяет проводить детализированный бенчмарк кода обработки исключений. Протестируем несколько возможных стратегий.

Первая стратегия — стандартный подход `try-catch`. Сравним ее с решением с монадой `Try`, в которую будет упакована причина исключения. Наконец, мы

увидим, как потребление трассировки стека отражается на производительности. Для потребления исключения будет использоваться стандартный вывод и регистрация `Throwable` в журнале.

Для начала потребуется эталонный метод, в котором не задействована обработка ошибок. По нему можно будет оценить, как исключения влияют на производительность. Каждая операция будет выполняться 50 000 раз для получения воспроизводимых результатов (однократное тестирование будет неинформативно). Для эмуляции этого поведения будет использован цикл `for`. Также вместо ручной реализации цикла `for` можно воспользоваться параметром `JMH`. Оба решения достаточно хорошо подходят для нашего сценария. В следующем листинге представлен эталонный метод оценки.

Листинг 3.31. Эталонный метод оценки исключений

```
private static final int NUMBER_OF_ITERATIONS = 50_000;
@Benchmark
public void baseline(Blackhole blackhole) {
    for (int i = 0; i < NUMBER_OF_ITERATIONS; i++) {
        blackhole.consume(new Object());
    }
}
```

Конструкция `JMH Blackhole` (см. <http://mng.bz/doVo>) моделирует реальное приращение проверочного кода. Если ее не использовать, появляется риск того, что JIT-компилятор оптимизирует или вообще удалит код. Проверочный код делает не так много — он только создает объект и позволяет `Blackhole` потратить его. Создадим первый проверочный код, как показано в следующем листинге. Выдадим исключение и перехватим его в блоке `catch`.

Листинг 3.32. Тестирование выдачи исключения

```
@Benchmark
public void throwCatch(Blackhole blackhole) {
    for (int i = 0; i < NUMBER_OF_ITERATIONS; i++) {
        try {
            throw new Exception();
        } catch (Exception e) {
            blackhole.consume(e);
        }
    }
}
```

Это позволяет оценить производительность стандартного кода обработки исключений. Обратите внимание на потребление исключения. Оно моделирует использование в реальном коде, но без анализа трассировки стека исключения или регистрации исключения в журнале. Следующий листинг показывает, как дополнить пакет тестов производительности этими операциями.

Листинг 3.33. Тестирование потребления трассировки стека

```

@Benchmark
public void getStackTrace(Blackhole blackhole) {
    for (int i = 0; i < NUMBER_OF_ITERATIONS; i++) {
        try {
            throw new Exception();
        } catch (Exception e) {
            blackhole.consume(e.getStackTrace());
        }
    }
}

@Benchmark
public void logError() {
    for (int i = 0; i < NUMBER_OF_ITERATIONS; i++) {
        try {
            throw new Exception();
        } catch (Exception e) {
            logger.error("Error", e);
        }
    }
}

```

Получает все трассировки стека, связанные с исключением

Передаёт исключение диспетчеру регистрации в журнале

Важно отметить, что при использовании диспетчера регистрации в журнале метод `error` получит трассировку стека, а также присоединит данные к журналу. Чтобы завершить набор, добавим бенчмарк-тест, который использует функциональный способ обработки сбоев, как показано в следующем листинге. Исключение будет упаковано в монаду `Try`, и объект `Try` должен потребляться программой.

Листинг 3.34. Тестирование монады `Try`

```

@Benchmark
public void tryMonad(Blackhole blackhole) {
    for (int i = 0; i < NUMBER_OF_ITERATIONS; i++) {
        blackhole.consume(Try.of(() -> { throw new Exception(); }));
    }
}

```

Упаковывает `Exception` в `Try` без обращения к трассировке стека

Рассмотрим результаты тестов производительности. Конкретные показатели могут различаться в зависимости от компьютера, но общая тенденция останется той же. На рис. 3.5 показаны результаты тестирования на моем компьютере.

Эталонная версия в среднем занимает менее одной миллисекунды (мс). Она соответствует коду, не включающему обработку исключений. Далее средняя операция `throwCatch` занимает менее 100 мс, и упаковка исключения в монаду `Try` дает практически идентичный результат. Это означает, что при выборе подхода (функционального или объектно-ориентированного) к обработке ошибок фактор производительности можно не учитывать. Ситуация становится более интересной, если проанализировать трассировку стека.

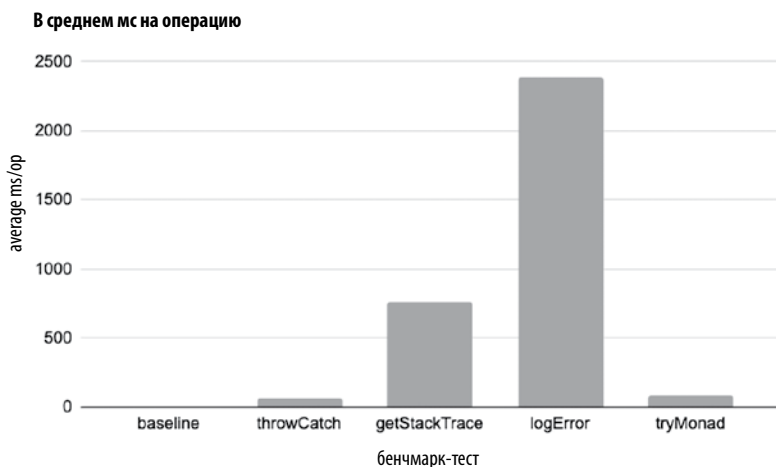


Рис. 3.5. Результаты тестирования исключений
(результаты могут различаться в зависимости от компьютера)

Если ограничиться получением трассировки стека (это означает, что создается и потребляется массив со всеми трассировками стека), код обработки исключения занимает около 750 мс на операцию. Получение трассировки стека выполняется почти в 10 раз медленнее, чем выдача и перехват исключения без анализа трассировки стека. Самой затратной процедурой является регистрация исключения в журнале. Она включает трассировку стека и построение строки сообщения на основе ее данных. Кроме того, она может включать логику присоединения, которая может задействовать операцию ввода/вывода для сохранения данных в файле на диске. Быстродействие регистрации исключения приблизительно в 30 раз ниже решения с `throw-catch` или с функциональной конструкцией `Try`. Также оно в три раза медленнее получения трассировки стека. Это выглядит разумно, поскольку требует значительных дополнительных усилий.

В завершение раздела, посвященного производительности, мы видим, что на практике работают и функциональный, и объектно-ориентированный подход к обработке ошибок — при условии, что анализировать трассировку стека не нужно. И даже если это необходимо, в большинстве случаев это нетрудно. Проблемы с производительностью могут появиться, если код злоупотребляет исключениями и выдает их практически в каждой программной ветви.

Конечно, в некоторых ситуациях потребуется распаковать трассировку стека исключения и зарегистрировать его для целей отладки. Показатели производительности в этом отношении информативны. Если перехватить исключение и выдать его заново, регистрация исключения на этом промежуточном шаге приведет к значительному ухудшению производительности. Если вы хотите заново

выдать исключение на более высокий уровень, его не следует регистрировать; оно будет обработано и зарегистрировано на этом уровне стека вызовов. Если вы перехватываете исключение без повторной выдачи, то одно выполнение, включающее трассировку стека, допустимо.

В большинстве сценариев использования (помимо высокочастотной обработки с низкой задержкой) последствия для производительности, рассмотренные в этом разделе, будут пренебрежимо малыми и их можно спокойно игнорировать. Таким образом, этот раздел следует рассматривать скорее как справочный, а не как повод отказаться от исключений там, где их использование обоснованно.

В этой главе были рассмотрены стратегии обработки исключений. Ошибки и исключения не должны использоваться для управления бизнес-логикой. Их цель — уведомить о непредвиденном поведении кода. Если не злоупотреблять исключениями, то проблем с производительностью, связанных с ними, не возникнет.

В следующей главе речь пойдет о том, как спрогнозировать необходимую пользователям функциональность. Вы увидите, что выгода от реализации некоторых новых возможностей сопровождается неоправданной сложностью и слишком большими затратами на их обслуживание.

ИТОГИ

- Иерархия исключений и ошибок существует во многих объектно-ориентированных языках. Для целей диагностики очень важно уметь в ней разбираться.
- При проектировании API обработки ошибок можно выбрать между проверяемыми и непроверяемыми исключениями. Проверяемые исключения являются явной частью таких API, а непроверяемые относятся к неявной части кода обработки ошибок; первые должны обрабатываться, для вторых это не обязательно.
- При проектировании логики обработки исключений для общедоступных API следует проанализировать достоинства и недостатки проверяемых и непроверяемых исключений и сравнить их с обработкой исключений в собственном коде.
- С API обработки исключений необходимо правильно реагировать на возникающие проблемы. Часто появляется соблазн тщательно проанализировать код и сделать вывод, что исключение не может быть выдано ни при каких обстоятельствах. Понимание распространенных антипаттернов логики обработки исключений поможет определиться в этом вопросе.
- При взаимодействии со сторонними библиотеками следует разработать стратегию обработки исключений. При интеграции со сторонними библио-

теками утечка типов исключений может привести к возникновению сильной связанности, поэтому важно понимать необходимость упаковки сторонних исключений.

- Обработку сбоев в асинхронных процессах с использованием многопоточности следует проводить с осторожностью; в противном случае возникает риск незаметных сбоев.
- Выдача исключений — не единственный способ обработки исключений в коде. Монада `Try` также инкапсулирует успех или сбой.
- Используя бенчмарк-тесты производительности для разных стратегий обработки исключений, можно определить, какие операции являются наиболее затратными.

4

Баланс между гибкостью и сложностью

https://t.me/it_boooks

В этой главе:

- ✓ Гибкость и расширяемость против затрат на обслуживание и сложность API.
- ✓ Обеспечение максимальной расширяемости с API перехватчиков и прослушивателей.
- ✓ Контроль над сложностью и защита от непрогнозируемого использования.

При проектировании систем и API нужен баланс между набором поддерживаемых возможностей и затратами на их обслуживание, которые проистекают из их сложности. В идеале каждое изменение API, такое как добавление новой функции, обосновано эмпирическими исследованиями. Например, можно проанализировать трафик на веб-сайте и, если нужно, добавить новый инструмент или воспользоваться A/B-тестированием (<http://mng.bz/ragJ>), чтобы решить, какие возможности оставить, а от каких отказаться. В зависимости от результатов A/B-тестирования можно избавиться от лишней функциональности.

Однако стоит заметить, что иногда удалить функцию из общедоступного API сложно или недопустимо. Например, если нужно сохранить обратную совместимость, исключение той или иной возможности может стать критическим

изменением, зачастую неприемлемым. Можно попытаться принудительно перевести клиенты на новый API без удаленных элементов, но это не просто. Тема совместимости более подробно рассматривается в главе 12.

При проектировании общедоступного API обычно лучше начинать с малого. Например, добавить ограниченный функционал, расширяя его на основании информации от конечных пользователей, а не реализовывать множество функций сразу без возможности их удаления в дальнейшем.

С другой стороны, при построении библиотек, которые будут использоваться другими специалистами и командами в организации, необходимо предусмотреть потребность в определенном функционале. Если создать библиотеку с минимальным набором возможностей и нерасширяемой архитектурой, может возникнуть ситуация, когда нам придется часто проводить рефакторинг и изменять API. С другой стороны, в попытке спрогнозировать все варианты использования кода можно создать слишком обширную кодовую базу, которая будет допускать расширение в любой точке, но сложность кода при этом резко возрастет. Эта глава поможет отыскать баланс между гибкостью и расширяемостью кодовой базы и сопутствующей сложностью и затратностью обслуживания.

4.1. МОЩНЫЙ, НО НЕ РАСШИРЯЕМЫЙ API

Предположим, ваша команда получила новое задание — создать программный компонент с общим доступом для других команд и клиентов. Это означает, что когда он будет написан, его будут использовать другие люди. Существует набор требований, которым должен удовлетворять код.

4.1.1. Проектирование нового компонента

В данном сценарии главная задача нового компонента — выполнение клиентами POST-запроса HTTP для заданного URL-адреса. Помимо этого, в код необходимо добавить метрики. Если запрос завершается успехом, увеличивается метрика `requests.success`. В случае сбоя должна расти метрика `requests.failure`.

Третья функциональность, которую должен предоставлять код, — возможность выполнить действие повторно. Вызывающая сторона задает максимальное количество повторных попыток. Если все они исчерпаны, обработка завершается неудачей. С другой стороны, если операция завершается успешно со второй попытки, эта попытка должна быть прозрачной для клиента. Необходимо, чтобы в коде увеличивалась метрика `requests.retry`, указывающая, что для выполнения запроса понадобились повторные попытки. Обратите внимание: сбой передается

на сторону клиента только после всех этих попыток. Можно построить диаграмму с набором функций, которые должен поддерживать API, как показано на рис. 4.1.

При проектировании такого компонента нужно отвечать на вопросы о сторонних библиотеках, которыми вы пользуетесь. Но что еще важнее, необходимо предвидеть сценарии использования, чтобы предусмотреть потенциальные точки расширения без чрезмерного усложнения нового компонента. Иногда разработчики склонны начинать реализацию, применяя паттерны, позволяющие расширить код в будущем. Тем самым они рискуют ввести много уровней абстракции, наращающих сложность системы.

В этой главе используется другой подход. Начнем мы с самой прямолинейной структуры без точек расширения. Затем перейдем к более гибкому с точки зрения клиента коду. При этом вы увидите, что гибкость также повышает сложность кода.

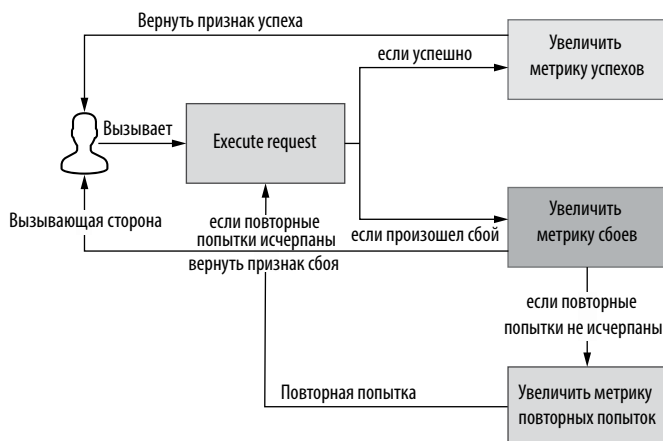


Рис. 4.1. Набор поддерживаемых функций совместно используемого программного компонента

4.1.2. Начиная с простого

Наше путешествие в область рефакторинга начнется с самой прямолинейной реализации. Сначала мы поймем, как она работает, а затем обсудим ее ограничения. Затем мы попробуем спрогнозировать отсутствующие сценарии использования и точки расширения, которые можно предоставить.

Назовем свой новый компонент `HttpClientExecution`. Конструктор этого компонента получает в аргументе `MetricRegistry` — класс сторонней библиотеки, используемый для работы с метриками (<https://metrics.dropwizard.io/4.2.0>). В следующем листинге представлена первая версия компонента.

Листинг 4.1. Параметры HttpClientExecution

```

import com.codahale.metrics.Meter;
import com.codahale.metrics.MetricRegistry;

private final int maxNumberOfRetries;
private final CloseableHttpClient client;
private final Meter successMeter;
private final Meter failureMeter;
private final Meter retryCounter;

public HttpClientExecution(
    MetricRegistry metricRegistry, int maxNumberOfRetries,
    CloseableHttpClient client) {
    this.successMeter = metricRegistry.meter("requests.success");
    this.failureMeter = metricRegistry.meter("requests.failure");
    this.retryCounter = metricRegistry.meter("requests.retry");
    this.maxNumberOfRetries = maxNumberOfRetries;
    this.client = client;
}

```

Устанавливает верхнюю границу для повторных попыток

Создает метрики с использованием MetricRegistry

Клиент предоставляется вызывающей стороной, которая отвечает за его настройку

В коде используется сторонняя библиотека, предоставляющая класс `MetricRegistry` (<http://mng.bz/Vlzy>). Он нужен для построения и публикации метрик из кода. Мы рассматриваем его по принципу «черного ящика» и используем его открытый API. Однако использование этого класса в компоненте привязывает `HttpClientExecution` к конкретной библиотеке метрик. Существует пара других библиотек метрик, и если клиент захочет переключиться на одну из них, код не позволит ему это сделать. Мы вернемся к этой проблеме позже.

А пока сосредоточимся на алгоритмической реализации выполнения с повторными попытками. Для обработки POST-запроса метод должен получать только один параметр — путь в формате `String`. Следующий листинг показывает, как реализуются метрики и как происходят повторные попытки.

Листинг 4.2. Выполнение POST с логикой повторных попыток

```

public void executeWithRetry(String path) {
    for (int i = 0; i <= maxNumberOfRetries; i++) {
        try {
            execute(path);
            return;
        } catch (IOException e) {
            logger.error("Проблема при отправке запроса о количестве повторных попыток: " + i, e);
            failureMeter.mark();
            if (maxNumberOfRetries == i) {
                logger.error("Это последняя попытка, и она завершилась сбоем.");
                throw new RuntimeException(e);
            } else {
                logger.info("Попробуйте снова.");
                retryCounter.mark();
            }
        }
    }
}

```

Продолжать, пока остаются повторные попытки

Если выполнение завершилось успешно, вернуть управление из метода

В случае сбоя увеличивается метрика сбоев

Если повторные попытки еще остались, увеличивает счетчик и продолжает выполнение логики

Если повторные попытки исчерпаны, упаковывает исключение и распространяет его на сторону вызова

```

    }
}

private void execute(String path) throws IOException {
    CloseableHttpResponse execute = client.execute(new HttpPost(path));
    if (execute.getStatusLine().getStatusCode() == HttpStatus.SC_OK) {
        successMeter.mark(); ← В случае успеха увеличивается метрика успехов
    } else {
        failureMeter.mark(); ← Любой код, кроме 200, считается сбоем
    }
}

```

Обратите внимание, что успехом считается только получение кода статуса HTTP 200. В любом другом случае — как для исключения, так и для кода статуса, отличного от 200, — увеличивается метрика сбоев `failureMeter`. Также можно внести изменение и рассматривать все коды 2xx как успех, но для нашего примера это несущественно.

Тщательно проанализировав этот алгоритм, можно заметить, что он реализует логику с рис. 4.1 с поддерживаемым функционалом. Наш код удовлетворяет требованиям, но не предоставляет возможностей для их расширения. Вызывающая сторона может влиять на его поведение только одним способом — передачей параметра `maxNumberOfRetries`. Позже мы изменим этот код и сделаем его более гибким, а эту версию можно использовать в качестве точки отсчета перед рефакторингом.

Наконец, чтобы понять сквозную логику, рассмотрим модульные тесты, оценивающие поведение этого компонента. Первый тест проверяет, что при успехе первой попытки выполняется только один запрос. В следующем листинге приведена реализация теста.

Листинг 4.3. Проверка успеха без повторных попыток

```

@Test
public void shouldNotRetryIfFirstRequestsSuccessful() throws IOException {
    // Дано
    MetricRegistry metricRegistry = new MetricRegistry();
    CloseableHttpClient client = mock(CloseableHttpClient.class);
    CloseableHttpResponse response = mock(CloseableHttpResponse.class);
    when(response.getStatusLine())
        .thenReturn(new BasicStatusLine(HTTP_1_1, HttpStatus.SC_OK, null));
    HttpClientExecution httpClientExecution = new
    HttpClientExecution(metricRegistry, 3, client);

    when(client.execute(any())).thenReturn(response); ← Моделирует клиент HTTP для возвращения успеха

    // Если
    httpClientExecution
    ➔ .executeWithRetry("http://localhost/user"); ← Выполняет метод executeWithRetry(), образующий открытый API компонента
}

```

```
// To
assertThat(getMetric(metricRegistry, "requests.success"))
➡ .isEqualTo(1);
assertThat(getMetric(metricRegistry, "requests.failure")).isEqualTo(0);
assertThat(getMetric(metricRegistry, "requests.retry")).isEqualTo(0);
```

Увеличивает метрику request.success

Если все последующие попытки завершаются неудачей, следует увеличить метрики сбоев и повторных попыток. Наконец, необходимо передать причину сбоя на сторону клиента, как показано в следующем листинге. В этом тесте моделируется сбой после всех повторных попыток.

Листинг 4.4. Проверка сбоя после повторных попыток

```
when(client.execute(any())).thenThrow(new IOException("problem"));
HttpClientExecution httpClientExecution = new
    HttpClientExecution(metricRegistry, 3, client);

// Если
assertThatThrownBy(
    () -> {
        httpClientExecution.executeWithRetry("url");
    })
    .hasCauseInstanceOf(IOException.class); ➡
// To
assertThat(getMetric(metricRegistry, "requests.success")).isEqualTo(0);
assertThat(getMetric(metricRegistry,
➡ "requests.failure")).isEqualTo(4);
assertThat(getMetric(metricRegistry,
➡ "requests.retry")).isEqualTo(3);
```

После всех повторных попыток распространяется исключение IOException

Задаёт три повторные попытки после первого запроса

Равно параметру, переданному HttpClientExecution

ПРИМЕЧАНИЕ

Значение метрики `requests.failure` будет больше количества повторных попыток клиента на 1. Это объясняется тем, что первый запрос не учитывается в общем количестве повторных попыток.

Наконец, если первый запрос завершается неудачей, но второй оказывается успешным, логика повторных попыток должна обеспечить прохождение вызова. С точки зрения клиента никакой информации о повторных попытках не будет. Только по метрикам клиент может узнать, были повторные попытки или нет. В следующем листинге приведен последний модульный тест, в котором первый запрос завершается неудачей, а второй успешен.

Листинг 4.5. Проверка сбоя с последующим успехом

```
when(client.execute(any())).thenThrow(new
    IOException("problem")).thenReturn(response); ➡
```

Моделирует сценарий со сбоем и успехом

```

HttpClientExecution httpClientExecution = new
    HttpClientExecution(metricRegistry, 3, client);

// Когда
httpClientExecution.executeWithRetry("url");

// Первый вызов завершился неудачей, после повторной попытки
// второй вызов был успешным.
assertThat(getMetric(metricRegistry,
    ➤ "requests.success")).isEqualTo(1);
assertThat(getMetric(metricRegistry,
    ➤ "requests.failure")).isEqualTo(1);
assertThat(getMetric(metricRegistry,
    ➤ "requests.retry")).isEqualTo(1)

```

Когда компонент завершит обработку,
должен быть один успех

Также должен быть один сбой:
первая попытка

Делается одна повторная
попытка, чтобы вторая была
успешной

Хотя компонент получается относительно незамысловатым и простым в обслуживании, он сопровождается рядом ограничений. Точек расширения не так много, и мы вынуждаем клиентский код использовать конкретную реализацию библиотеки метрик. Так образуется сильная связанность между нашим компонентом и сторонней библиотекой.

Допустим, вы решили усовершенствовать компонент и повысить его гибкость. Конечный пользователь должен иметь возможность выбрать свою библиотеку и предоставить реализацию. Для нашего компонента не имеет значения, какая именно библиотека используется для сбора метрик. В следующем разделе вы увидите, как попытки предвидеть это улучшение повлияют на кодовую базу.

4.2. ВОЗМОЖНОСТЬ ПРЕДОСТАВЛЕНИЯ СОБСТВЕННОЙ БИБЛИОТЕКИ МЕТРИК

Пока что наш компонент не очень гибкий. В нем существует жесткая зависимость от сторонней библиотеки, отвечающей за сбор метрик. На первый взгляд это не создает проблем, но другие разработчики и системы будут пользоваться нашим кодом. Применяя любой класс из сторонней библиотеки в кодовой базе, мы ограничиваем будущие реализации своего компонента. Более того, тем самым мы устанавливаем ограничение, требующее, чтобы в каждой точке, в которой задействуется код, использовалась одна и та же библиотека метрик.

Взглянув на директивы импортирования в компоненте `HttpClientExecution`, можно заметить зависимости от сторонних библиотек. Эти зависимости представлены в следующем листинге.

Листинг 4.6. Зависимости от сторонних библиотек

```
import com.codahale.metrics.Meter;
import com.codahale.metrics.MetricRegistry;
```

Выясняется, что наш простой код содержит сильную связанность, которая усложняет его тестирование и расширение. По этой причине мы абстрагируем код, относящийся к метрикам. Паттерн абстрагирования достаточно прост: необходимо определить обобщенный интерфейс, который может стать точкой входа для системы (рис. 4.2). В дальнейшем код будет интегрироваться с любой конкретной реализацией только через новый интерфейс.

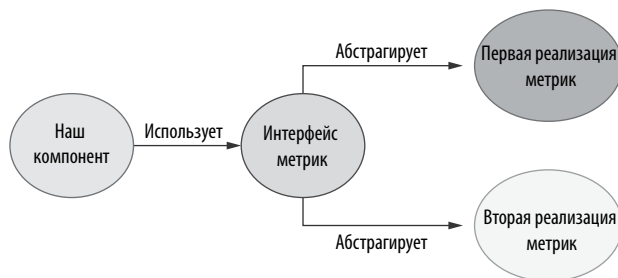


Рис. 4.2. Абстрагирование сторонней библиотеки метрик с использованием интерфейса

Новый интерфейс метрик должен определить контракт между компонентом и сторонними библиотеками. Он может быть очень простым (см. следующий листинг).

Листинг 4.7. Определение интерфейса метрик

```
public interface MetricsProvider {
    void incrementSuccess();
    void incrementFailure();
    void incrementRetry();
}
```

Этот интерфейс не позволяет вызывающей стороне получить данные из компонента. Это довольно серьезное ограничение, потому что вызывающая сторона может внедрить собственный реестр для отслеживания метрик. Вследствие этого вызывающая сторона владеет кодом реестра метрик и может обращаться к метрикам напрямую. Нет необходимости включать в интерфейс `MetricsProvider` методы доступа. И `MetricsProvider` не должен импортировать никакие классы из сторонних библиотек метрик, так что между `MetricsProvider` и конкретной реализацией метрик не существует сильной связанности.

`HttpClientExecution` взаимодействует с метриками через эту абстракцию. При этом нам не нужно беспокоиться о подробностях реализации — клиент

предоставляет их. Допустим, клиент хочет обеспечить реализацию для библиотеки Dropwizard. Что еще важнее, он должен внедрить интерфейс `MetricsProvider`. Реализация приведена в следующем листинге.

Листинг 4.8. Реализация провайдера метрик

```
public class DefaultMetricsProvider implements MetricsProvider {
    private final Meter successMeter;
    private final Meter failureMeter;
    private final Meter retryCounter;

    public DefaultMetricsProvider(MetricRegistry metricRegistry) {
        this.successMeter =
        ➤ metricRegistry.meter("requests.success");
        this.failureMeter = metricRegistry.meter("requests.failure");
        this.retryCounter = metricRegistry.meter("requests.retry");
    }

    @Override
    public void incrementSuccess() {
        successMeter.mark();
    }

    @Override
    public void incrementFailure() {
        failureMeter.mark();
    }

    @Override
    public void incrementRetry() {
        retryCounter.mark();
    }
}
```

Эта реализация предоставляет внутренние подробности

Методы интерфейса становятся единственными точками интеграции для компонента

Как прогнозирование этой функции повлияло на гибкость и сложность компонента? Во-первых, подробности реализации были абстрагированы из компонента, а логика `HttpClientExecution` упростилась. Теперь нет необходимости внедрять эти подробности в систему. Отсюда следует однозначный вывод: сложность компонента снизилась. А поскольку клиенты теперь могут предоставить любую реализацию метрик по своему усмотрению, гибкость также увеличилась.

Казалось бы, повышение гибкости и сокращение сложности можно считать однозначным выигрышем. Тем не менее у этой медали есть и обратная сторона. Сложность, удаленная из системы, должна где-то находиться. Если несколько клиентов используют наш компонент, всем им придется реализовать новый интерфейс метрик. Теперь сложность находится в различных клиентских репозиториях. По сути, она была вынесена на сторону внешних клиентов. Гибкость повысилась, но при этом увеличилась сложность — просто не в кодовой базе.

С точки зрения клиента необходимость совершать множество дополнительных шагов при использовании компонента — например, предоставление собственной реализации метрик — может быть трудозатратной, и клиент в итоге выберет другую систему или компонент. Хорошим промежуточным решением может стать выделение интерфейса метрик с предоставлением реализации по умолчанию, которая подходит для большинства пользователей. Тем самым обеспечивается расширяемость, но сложность перемещается в систему. Она не выносится в клиентский код. Если клиент захочет предоставить другую реализацию, он легко реализует нужную функциональность и предоставит ее нашему компоненту.

В реальной жизни системы зависят от множества внешних компонентов, и иногда создать абстракцию невозможно. Даже в нашем простом примере существует зависимость от фактической реализации клиента HTTP. Клиент HTTP предоставляет другие методы, которые может быть трудно скрыть от абстракции. Разработка абстракции, скрывающей клиент HTTP, с попыткой предвидеть все возможные сценарии использования такой библиотеки повысит сложность кода.

В следующем разделе рассмотрен один из самых гибких и расширяемых механизмов для предоставления точек расширения в случае, если известно, что сложность архитектуры значительно возрастет. Мы рассмотрим механизм API перехватчиков, позволяющий клиентам предоставлять свое поведение в различных точках жизненного цикла компонента. В примерах задействован другой сценарий использования (который не имеет отношения к метрикам).

4.3. ОБЕСПЕЧЕНИЕ РАСШИРЯЕМОСТИ API С ИСПОЛЬЗОВАНИЕМ ПЕРЕХВАТЧИКОВ

У каждого фреймворка и системы существует жизненный цикл, состоящий из нескольких этапов. Вместо того чтобы пытаться предвидеть все возможные сценарии использования для клиентов, можно разрешить клиентам предоставлять нужное поведение и внедрять его в компонент. Тогда нам больше не придется менять свой API и код на основании новых запросов функций. Клиенты могут предоставлять собственную логику, и для кода это не должно иметь значения. Теоретически такой подход значительно улучшает расширяемость, и о ней не придется беспокоиться. На практике код в систему необходимо внедрять осторожно. О нем трудно что-либо предположить, потому что мы ничего не знаем о поведении, предоставляемом вызываемой стороной.

Каждую фазу жизненного цикла можно сделать расширяемой и гибкой с применением различных паттернов, включая абстрагирование (описанное в предыдущем разделе) и наследование. Когда требуется обеспечить максимальную гибкость кода для клиентов, часто используется механизм перехватчиков (hooks). Этот паттерн позволяет клиентам подключать собственный код между фазами жизненного

цикла конкретного компонента. В API из примера нужно разрешить клиентам подключать код после подготовки запроса HTTP, но перед отправкой фактического запроса HTTP конечной точке REST. Эта схема изображена на рис. 4.3.

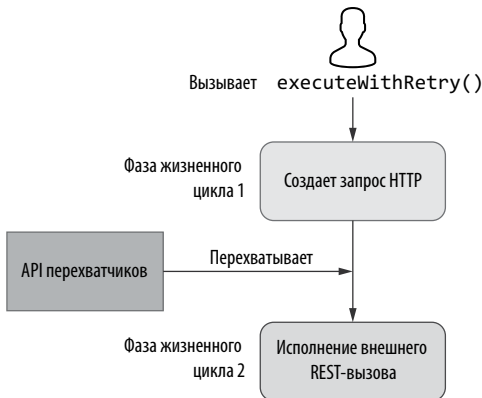


Рис. 4.3. Интеграция API перехватчиков с кодовой базой

Сначала клиент выполняет метод `executeWithRetry()`. Он запускает жизненный цикл компонента, который создает метод запроса HTTP. Обычно после завершения этой фазы жизненного цикла метод выполняет внешний REST-вызов, и цикл завершается. Добавление API перехватчиков разрешает клиентам *перехватывать* конкретный вызов. Клиенту остается только реализовать интерфейс перехватчика.

Затем перехватчик вызывается в соответствующем жизненном цикле компонента. Так обеспечивается гибкий механизм, избавляющий от бремени предвидения конкретных функций, которые могут понадобиться клиентам в будущем.

Первый шаг поддержки API перехватчиков — создание интерфейса, позволяющего коду вызвать перехватчик в конкретной фазе жизненного цикла кода. Новый интерфейс очень прост: он состоит всего из одного метода, как показывает следующий листинг. Мы вызываем этот метод, передавая `HttpRequestBase` в аргументе, который создается в первой фазе цикла.

Листинг 4.9. Реализация интерфейса перехватчиков

```
public interface HttpRequestHook {
    void executeOnRequest(HttpRequestBase httpRequest);
}
```

Клиент компонента внедряет реализацию перехватчика (клиенты могут внедрять сразу несколько). Следовательно, конструктор должен получать список перехватчиков, как показывает следующий листинг.

Листинг 4.10. Использование конструктора перехватчиков

```

public HttpClientExecution(
    MetricRegistry metricRegistry,
    int maxNumberOfRetries,
    CloseableHttpClient client,
    List<HttpRequestHook> httpRequestHooks) {
this.metricRegistry = metricRegistry;
this.successMeter = metricRegistry.meter("requests.success");
this.failureMeter = metricRegistry.meter("requests.failure");
this.retryCounter = metricRegistry.meter("requests.retry");
this.maxNumberOfRetries = maxNumberOfRetries;
this.client = client;
this.httpRequestHooks = httpRequestHooks;
}

```

Код клиента внедряет перехватчики

Необходимо сохранить для использования в будущем

Теперь разберемся, как API перехватчиков подключается к существующему жизненному циклу компонента. Так как код клиента выполняется между фазами цикла, он перебирает все внедренные перехватчики и передает объект `HttpPost` вызывающему коду. Схема показана в следующем листинге.

Листинг 4.11. Выполнение без обработки ошибок

```

private void execute(String path) throws IOException {
    HttpPost httpPost = new HttpPost(path);
    for (HttpRequestHook httpRequestHook : httpRequestHooks) {
        httpRequestHook.executeOnRequest(httpPost);
    }
    CloseableHttpResponse execute = client.execute(httpPost);
    if (execute.getStatusLine().getStatusCode() == HttpStatus.SC_OK) {
        successMeter.mark();
    }
}

```

В первой фазе жизненного цикла создается объект `HttpPost`

Выполняет клиентский код между фазами жизненного цикла

Вторая фаза жизненного цикла выполняет внешний вызов REST

Этот механизм позволяет клиентам внедрять свой код в промежуточных стадиях обработки. Здесь клиентскому коду передается POST-запрос HTTP, с которым вызывающая сторона может выполнить любое действие. Таким образом, данное решение обладает исключительно высокой гибкостью.

4.3.1. Защита от непредвиденного использования API перехватчиков

В этом примере перебираются все перехватчики, внедренные клиентом, после чего метод `HttpPost()` передается API. В этой точке можно заключить, что высокая расширяемость достигнута без существенного усложнения кода.

К сожалению, следует понимать, что на код клиента повлиять невозможно. Интерфейс перехватчиков не объявляет исключений. Но как вы узнали в главе 3, клиенты все равно могут выдавать непроверяемые исключения из своего кода. Это означает, что непредвиденное поведение в клиентском коде может привести к непроверяемому исключению.

ДОКУМЕНТИРОВАНИЕ КОНТРАКТОВ ПЕРЕХВАТЧИКОВ, НЕ ВЫДАЮЩИХ ИСКЛЮЧЕНИЯ, И ЗАЩИТА ОТ ПЕРЕХВАТЧИКОВ, КОТОРЫЕ МОГУТ ИХ ВЫДАВАТЬ

В идеальном мире разработчик, предоставляющий клиентам пользовательский API, должен документировать его контракт. Например, можно объявить, что все реализации перехватчиков не должны выдавать исключения. Тем не менее установить это требование для всех клиентов достаточно трудно. Кто-то непременно забудет (или не прочитает документацию), кто-то зависит от другого кода, который выдает непроверяемые исключения, хотя и не должен этого делать. А значит, если вы утверждаете, что исключения выдаваться не должны, разумно защищаться от любой возможности их появления. В противном случае в приложении, использующем ваш код, могут возникать трудновывяемые ошибки (или незаметные сбои).

Чтобы проверить это предположение, напишите модульный тест с перехватчиком, выдающим непроверяемое исключение. Пример теста представлен в следующем листинге.

Листинг 4.12. Тестирование непредвиденных ошибок в перехватчике

```
HttpClientExecution httpClientExecution =
    new HttpClientExecution(
        metricRegistry,
        3,
        client,
        Collections.singletonList(
            httpRequest -> {
                throw new RuntimeException("Непредвиденная ошибка!");
            }
        )
    );
```

Непредвиденная ошибка выдается в коде, который вы не контролируете

Такая ситуация повлияет на жизненный цикл компонента; обеспечивая гибкость для клиента, мы вводим сложность в код. Чтобы защититься от подобных проблем, необходимо упаковать код, который нам не принадлежит, в блок try-catch, как показано в следующем листинге.

Листинг 4.13. Защита от сбоев

```
for (HttpRequestHook httpRequestHook : httpRequestHooks) {
    try {
        httpRequestHook.executeOnRequest(httpPost);
    } catch (Exception ex) {
        logger.error("HttpRequestHook выдал исключение. Проверьте логику перехватчика", ex);
    }
}
```

Может произойти что угодно, поэтому может быть выдана любая разновидность исключений

Выдача исключения клиентом не должна завершать жизненный цикл компонента

Ошибку можно зарегистрировать, чтобы предоставить обратную связь клиенту в том случае, если исключение не было критическим с точки зрения базовой бизнес-обработки (кода, вызывающего перехватчики). Другими словами, независимо от типа логики, внедряемой вызывающей стороной, эти проблемы не должны влиять на обработку. Если в коде, предоставленном перехватчиком, происходит сбой, его можно зарегистрировать для отладки, но выполнение все равно может продолжиться.

Если разрешить дальнейшее распространение исключения, это повлияет на логику библиотеки. Делать этого не следует, поскольку код библиотеки работает согласно ожиданиям, но ситуация может еще сильнее усложниться. Если передать API перехватчиков объект с состоянием — такой, как клиентский объект HTTP, — вы не сможете повлиять на то, как он будет использоваться. Код API перехватчиков способен выполнять код, влияющий на внутренние механизмы клиента. Например, он может быть использован для выполнения дополнительных запросов HTTP. Это создаст проблемы, если вы оптимизировали клиент HTTP для своего трафика (настроили размер очереди, тайм-ауты и другие параметры).

Логика, предоставленная компоненту, потребляет ресурсы, необходимые для ее нормальной работы. Это может привести к нарушению SLA сервиса и даже всего жизненного цикла. В худшем случае код клиента выполнит логику, которая приведет к сбою клиента HTTP. Тогда в вашем компоненте также произойдет сбой.

ПРИМЕЧАНИЕ

При передаче любого внутреннего состояния коду, который вам не принадлежит, необходимо действовать осторожно. Документирование предположений API перехватчиков может стать хорошим первым шагом, но оно не предотвратит непредвиденное использование в реальной жизни.

В этом разделе вы узнали, как защититься от непредвиденного использования с точки зрения правильности кода. А что насчет производительности? Эта проблема рассматривается ниже.

4.3.2. Влияние API перехватчиков на производительность

С точки зрения правильности наш код должен обрабатывать непредвиденные сбои в клиентском коде. Однако следует понимать, что логика, предоставленная клиентом, может быть блокирующей. А это значит, что на время, которое занимает вызов API перехватчиков, повлиять нельзя.

Предположим, вызывающая сторона предоставляет логику перехватчиков, которая выполняет функцию ввода/вывода — например, сетевую функцию или функцию файловой системы, подразумевающую блокирование. Каждый вызов ввода/вывода может быть непредсказуемым и может привести к более высокой

задержке. Допустим, задержка вызова API перехватчиков составляет 1000 мс. Если первая фаза жизненного цикла занимает 100 мс, а вторая — 200 мс, общее время одного вызова составит 1300 мс вместо 300. В четыре раза медленнее! Это значительно влияет на производительность компонента. На рис. 4.4 представлена эта ситуация.

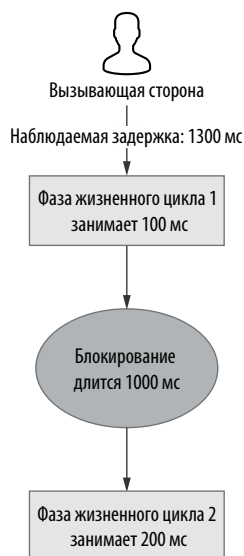


Рис. 4.4. Блокирующий вызов API перехватчиков

Вызывающая сторона заметит высокую задержку, и при текущей структуре компонента сделать с этим ничего не удастся. Синхронный вызов внутри перехватчика использует тот же поток, что и наш компонент, из-за чего в некоторых ситуациях даже возникает риск взаимной блокировки.

Представьте сценарий, в котором имеется минимальное количество потоков, а клиентский код, предоставленный в API перехватчиков, блокируется или ожидает освобождения некоторого внешнего ресурса. Даже если ресурс будет недоступен хотя бы один раз, поток, совместно используемый нашим компонентом и клиентским кодом, заблокируется. Если это произойдет несколько раз, может возникнуть проблема нехватки потоков для обработки запросов.

Можно потребовать, чтобы клиенты выполняли неблокирующие вызовы в коде обратного вызова. Это означает, что каждому клиенту придется управлять пулом потоков, которые будут управлять действиями API перехватчиков. И снова, даже если документировать это требование, невозможно легко обеспечить его соблюдение. Блокирующий вызов можно обнаружить в потоке, но это существенно усложнит структуру.

Ситуация усугубляется, если имеются несколько независимых действий перехватчиков и каждый из них выполняет логику, которая подразумевает блокирующее поведение. Если есть два блокирующих перехватчика и каждый из них занимает 1000 мс, общее время обработки возрастает до 2300 мс, что в 8 раз медленнее кода без перехватчиков (рис. 4.5).

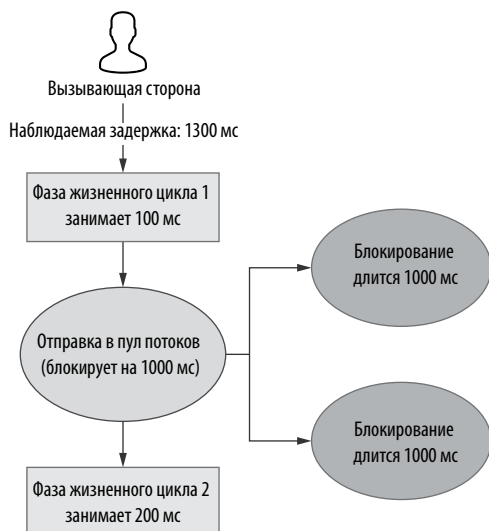


Рис. 4.5. API перехватчиков распараллеливает блокирующие вызовы

Одно из возможных решений — считать, что код API перехватчиков небезопасен и всегда может быть блокирующим. Вводя это утверждение, можно упаковать каждый вызов API перехватчиков в действие, выполняемое в отдельном потоке.

С текущей архитектурой каждый API перехватчиков передается отдельному пулу потоков, причем управление и обслуживание пула происходит в нашем коде. Нужно определиться с количеством необходимых потоков, размером очереди и другими факторами (например, разрешать динамическое добавление потоков или нет). Также мы должны наблюдать за использованием пула и потреблением потоков.

Управление отдельным пулом потоков увеличивает сложность кодовой базы. Пул содержит очередь задач, которые могут обрабатываться, и нужно контролировать эту очередь, чтобы избежать риска исчерпания памяти из-за незавершенных задач. Кроме того, необходимо проследить, чтобы потоки не уничтожались незаметно при возникновении непредвиденных исключений.

Можно отправить n параллельных выполнений перехватчиков, где n — количество потоков в пуле. Допустим, вызывающая сторона предоставила два

перехватчика. Каждый из них выполняет блокирующий вызов, который продолжается 1000 мс. В идеале эти перехватчики должны выполняться по отдельности, без ожидания между стадиями. В этом случае они не повлияют на задержку вызова компонента. Но как вы помните, новый API позволяет клиентам подключать свой код между фазами жизненного цикла. Из-за этого возникает отношение типа «*происходит до*» между завершением всех вызовов API перехватчиков и переходом к следующей фазе цикла. Даже если вызовы удастся распараллелить, необходимо дождаться, когда все они завершатся. Таким образом, добавляемая задержка будет как минимум не меньше времени выполнения самой медленной операции, предоставляемой через API перехватчиков. Гибкость архитектуры приводит к снижению производительности из-за увеличения задержки.

Рассмотрим структуру компонента `HttpClientExecution`, в которой учитываются улучшения, относящиеся к правильности и производительности. Необходимо создать специализированный пул потоков для API перехватчиков, что приводит к увеличению сложности и затрат на обслуживание. В следующем листинге каждый вызов API перехватчиков передается пулу потоков.

Листинг 4.14. Улучшение параллелизма

```
private final ExecutorService executorService =
    Executors.newFixedThreadPool(8);
private void executeWithErrorHandlingAndParallel(String path) throws
    Exception {
    HttpPost httpPost = new HttpPost(path);
    List<Callable<Object>> tasks =
    ➔ new ArrayList<>(httpRequestHooks.size());
    for (HttpRequestHook httpRequestHook : httpRequestHooks) {
        tasks.add(
            Executors.callable(
                () -> {
                    try {
                        httpRequestHook.executeOnRequest(httpPost);
                    } catch (Exception ex) {
                        logger.error(
                            "HttpRequestHook выдал исключение. Проверьте
                            логику перехватчика", ex);
                    }
                })
        );
    }
    List<Future<Object>> responses =
    ➔ executorService.invokeAll(tasks);
    for (Future<Object> response : responses) {
        response.get();
    }

    CloseableHttpResponse execute = client.execute(httpPost);
    if (execute.getStatusLine().getStatusCode() == HttpStatus.SC_OK) {
        successMeter.mark();
    }
}
```

Требуется задач, где количество равно количеству перехватчиков

Конструирует callable для каждого действия перехватчика

Активизирует все задачи

Перебор списка незавершенных задач

Ожидание каждого асинхронного действия перед переходом к следующему шагу

Последняя стадия выполняется после завершения всех перехватчиков

По сравнению с первой версией компонента код усложняется. Необходимо обработать все сбои, чтобы не рисковать незаметным уничтожением потоков в пуле. Кроме того, выполнение перехватчиков необходимо распараллеливать, но это не решает всех возникающих из-за них проблем производительности. Все равно нужно дождаться их завершения. Гибкость, которая достигается при использовании клиентами API перехватчиков, не дается просто так. За нее приходится расплачиваться повышением затрат на обслуживание кода управления пулами потоков и усложнением решения.

В следующем разделе рассмотрен другой механизм — API прослушивателей. Он позволяет улучшить гибкость API без необходимости прогноза всех возможных запросов функций от клиентов. Также мы реализуем передачу сведений о количестве повторных попыток на сторону клиента. Наконец, вы увидите, как проведение этой реализации на более поздней стадии приводит к усложнению логики.

4.4. ОБЕСПЕЧЕНИЕ РАСШИРЯЕМОСТИ API ЗА СЧЕТ ИСПОЛЬЗОВАНИЯ ПРОСЛУШИВАТЕЛЕЙ

На первый взгляд может показаться, что API прослушивателей похож на API перехватчиков, однако между ними есть различия, которые следует объяснить отдельно. Возможно, вы помните, что архитектура API перехватчиков синхронна, поскольку нужно дождаться завершения всех перехватчиков перед переходом на следующий шаг. Как показано на рис. 4.6, паттерн Наблюдатель (Observer), предоставляющий API прослушивателей, использует другой подход к созданию точек расширения для клиента. Наш компонент (в паттерне Наблюдатель он называется *субъектом*) позволяет клиентам регистрировать наблюдатели. Зарегистрированные наблюдатели получают уведомления при возникновении события в компоненте.

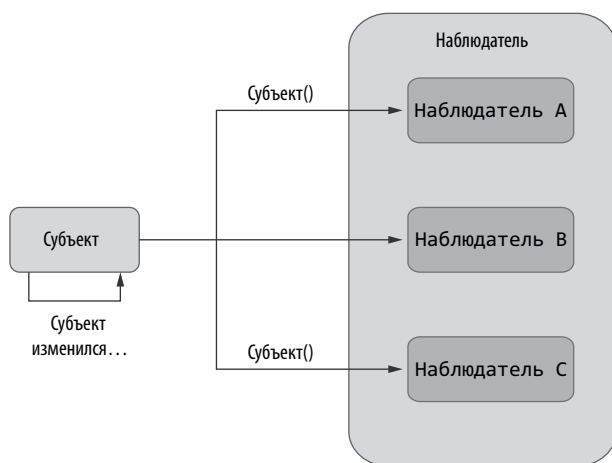


Рис. 4.6. Паттерн проектирования Наблюдатель позволяет клиентам регистрировать наблюдатели

Клиент может предоставить несколько наблюдателей. Рассмотрим наиболее значимые отличия между API прослушивателей и API перехватчиков.

4.4.1. Прослушиватели и перехватчики

При отправке события (например, обозначающего, что компонент завершил фазу жизненного цикла) уведомление происходит полностью асинхронно. Никаких отношений типа «происходит до» между событиями и переходом к следующей стадии компонента не происходит. Это означает, что при условии выполнения прослушивателей в отдельном пуле потоков риск снижения производительности отсутствует. Единственное отличие в том, что не нужно ожидать завершения действий API прослушивателя.

Идея предоставить доступ к внутреннему состоянию или сигнализировать о возникновении некоторого события с использованием API прослушивателей выглядит соблазнительно. Это гибкая абстракция, ведь можно разрешить клиентам предоставить собственное поведение без изменения нашего кода или API. Допустим, вы решили, что можете предвидеть новый сценарий использования, который требует, чтобы при завершении выполнения компонента отправлялось уведомление с состоянием повторных попыток (рис. 4.7).

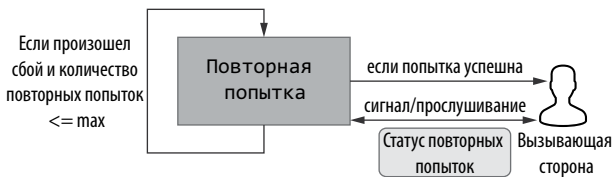


Рис. 4.7. Отправка состояния повторных попыток с использованием API прослушивателей

Информация о повторных попытках раскрывается после завершения всех попыток. Для этой цели создан новый класс `RetryStatus`, инкапсулирующий информацию, которая распространяется на сторону клиента. Новый класс представлен в следующем листинге.

Листинг 4.15. Создание класса `RetryStatus`

```
public class RetryStatus {
    private final Integer retryNumber;
    public RetryStatus(Integer retryNumber) {
        this.retryNumber = retryNumber;
    }
    public Integer getRetryNumber() {
        return retryNumber;
    }
}
```

Для простоты в этом примере статус содержит только количество повторных попыток. Он возвращает счетчик для конкретной повторной попытки.

Клиент может зарегистрировать прослушиватель `OnRetryListener`, который вызывается компонентом при возникновении ожидаемого действия. Интерфейс `OnRetryListener` содержит единственный метод `onRetry()` для получения статуса повторных попыток, и каждая такая попытка имеет собственный объект `RetryStatus`. Реализация приведена в следующем листинге.

Листинг 4.16. `OnRetryListener`

```
public interface OnRetryListener {
    void onRetry(List<RetryStatus> retryStatus);
}
public class HttpClientExecution {
    private final List<OnRetryListener> retryListeners = new ArrayList<>();
    public void registerOnRetryListener(OnRetryListener onRetryListener) {
        retryListeners.add(onRetryListener);
    }
}
// Остальные методы
}
```

Хранит список прослушивателей

Регистрирует новый прослушиватель при помощи специального метода

Когда логика повторных попыток завершится, можно перебрать все прослушиватели и распространить все статусы повторных попыток для заданного исключения. Их нужно объединить в список, который затем отправляется в каждый прослушиватель повторных попыток. В следующем листинге показана эта реализация.

Листинг 4.17. Вызов `OnRetryListener`

```
public void executeWithRetry(String path) {
    List<RetryStatus> retryStatuses = new ArrayList<>();
    for (int i = 0; i <= maxNumberOfRetries; i++) {
        // Логика повторных попыток
        retryListeners.forEach(1 -> 1.onRetry(retryStatuses));
    }
}
```

Объединяет все объекты `RetryStatus` в список

Распространяет статусы повторных попыток на все прослушиватели

Реальная логика повторных попыток остается неизменной

Заметьте, что при обработке сбоев и параллельного выполнения действуют те же правила, которые обсуждались в разделе 4.3. На первый взгляд логика кажется верной, но при распространении внутреннего состояния на код вызывающей стороны нужно учитывать, что невозможно помешать вызывающей стороне изменить переданное состояние. Изменение состояния может нарушить логику выполнения. Эта проблема будет рассмотрена в следующем разделе.

4.4.2. Неизменяемость архитектуры

При распространении любого состояния из компонента нельзя быть уверенными в том, что в клиентском коде оно останется неизменным. В следующем листинге

представлен модульный тест, который можно создать для моделирования этого поведения.

Листинг 4.18. Изменение состояния через прослушиватель

```
httpClientExecution.registerOnRetryListener(
    List::clear);
httpClientExecution.registerOnRetryListener(
    statuses -> {
        assertThat(statuses.size()).isEqualTo(1);
    });
```

← Клиент сбрасывает или изменяет статус, что приводит к побочному эффекту!

← Должен быть только один статус повторной попытки

Список статусов повторных попыток, переданный `OnRetryListener`, представляет собой ссылку на реальный список. Ничто не мешает клиенту вызвать `clear()`, удалить или добавить элемент в этот список. Если первый прослушиватель очистит статусы, то второй не увидит изменений. Это значит, что код вызывающей стороны создает побочный эффект, вследствие чего API становится недетерминированным, а риск ошибок возрастает. Чтобы избежать этого, можно создать копию реального списка, которая передается клиенту. Этот прием показан в следующем листинге.

Листинг 4.19. Копирование объекта, распространяемого на сторону прослушивателей

```
retryListeners.forEach(1 -> 1.onRetry(new ArrayList<>(retryStatuses)));
```

Для каждого прослушивателя создается копия статусов повторных попыток, которая отправляется всем прослушивателям. Даже если вызывающая сторона изменит объект, это не повлияет на другие прослушиватели кода API. Тем не менее у такого подхода есть недостатки.

Во-первых, при *глубоком копировании* исходного объекта придется создать множество копий данных. В этом случае все значения копируются из исходного в новый объект, что существенно повышает затраты памяти приложения. Для n прослушивателей реальные данные придется скопировать n раз, что повысит затраты памяти на соответствующую величину. Во-вторых, в коде прослушивателя возникает вероятность незаметных сбоев. Клиент может изменить список непреднамеренно, и иногда лучше явно сигнализировать о том, что такие операции запрещены, выдав исключение.

Обе проблемы можно решить упаковкой статуса в неизменяемую обертку. Для списка можно воспользоваться конструкцией `ImmutableList` (<http://mng.bz/xvzd>), как показано в следующем листинге.

Листинг 4.20. Оборачивание в неизменяемый объект

```
retryListeners.forEach(1 -> 1.onRetry(ImmutableList.copyOf(retryStatuses)));
```

Оборачивание реального состояния в неизменяемую абстракцию не приводит к созданию копии. Это создает класс, который выдает исключение при любых попытках изменения используемого списка. Отпадает необходимость в копировании реального содержимого списка каждый раз, когда оно распространяется на сторону прослушивателя. Запрещены только методы, вносящие изменения. Второе преимущество такого решения — явное выражение и обеспечение быстрого сбоя. Если код в API прослушивателя случайно меняет содержимое списка, то обратная связь предоставляется немедленно, а риск незаметного сбоя отсутствует.

Если вы распространяете любое состояние в код, который вам не принадлежит, всегда следует исходить из того, что оно неизменяемо. Можно сделать его таковым изначально за счет соответствующей архитектуры, применив неизменяемые классы и финальные поля. Если используемый API неизменяем, создавать защитную копию не обязательно. Затраты памяти при этом заметно снижаются.

В реальности часто приходится использовать библиотеки, не обеспечивающие неизменяемость. Многие коллекции — списки, множества и т. д. — изменяемы. Распространять их на сторону клиента следует очень осторожно. В таком случае состояние упаковывается в неизменяемый класс, который скрывает или запрещает модификацию используемых данных.

Другая проблема заключается в том, что распространение состояния в API прослушивателей создает риск их перегрузки трафиком. Если при каждом действии будут активизироваться n прослушивателей, потребление памяти приложением заметно возрастет. Есть риск, что код вызывающей стороны не справится с трафиком и будет заблокирован, что повлияет на основную функцию приложения. Подумайте о введении обратного давления (back pressure) или буферизации дополнительных событий статуса и их пакетной отправки. Любой вариант весьма усложнит архитектуру. Если вы решите отправлять уведомления из своего компонента, будьте внимательны.

Как видите, даже простое распространение состояния с использованием API прослушивателей усложняет код: необходимо следить, чтобы данные были неизменяемыми и чтобы код вызывающей стороны справлялся с трафиком. Попытки спрогнозировать сценарии использования на первый взгляд кажутся простыми, однако они значительно повышают сложность кода.

4.5. АНАЛИЗ ГИБКОСТИ API И ЗАТРАТЫ НА ОБСЛУЖИВАНИЕ

Самый главный вывод из примеров этой главы — каждая новая функция в той или иной степени повышает сложность. Например, иногда требуется

абстрагировать конкретную библиотеку, от которой зависит код. Подобный пример уже встречался при абстрагировании библиотеки метрик. Абстрагирование реализации снижает сложность в компоненте, но повышает ее в коде клиента. Каждый клиент должен предоставить реализацию для конкретной библиотеки метрик. По нашему опыту, для большинства сценариев использования оптимально гибридное решение — абстрагирование с предоставлением реализации по умолчанию, которая применяется чаще других.

Если попытаться предугадать конкретный сценарий использования, может возникнуть искушение включить относительно общие паттерны — такие, как API прослушивателей или перехватчиков. На первый взгляд они добавляют гибкости без значительного усложнения. И возможно, это действительно так, но за расширяемость приходится расплачиваться повышением сложности.

Применяя API перехватчиков, необходимо защититься от любого непредвиденного использования. Это значит, что код должен быть готов к любому исключению. Кроме того, нужно иметь в виду потоковую модель выполнения точек расширения API. Если архитектура допускает синхронные клиентские вызовы, следует исходить из того, что у некоторых клиентов они будут блокироваться, что повлияет на условия SLA компонента и использование ресурсов (например, потоков). Однако введение асинхронной логики, которая должна работать в коде параллельно, создает дополнительную сложность. Придется поддерживать специализированный пул потоков и отслеживать его работу.

Также необходимо обратить внимание на точки расширения компонента и отношение «происходит до» между фазами обработки. Если эти отношения существуют, то даже переход на параллельную обработку не снизит дополнительную задержку до нуля. Это сложность, которая неизбежно возникает при предоставлении API перехватчиков вызывающим сторонам.

API прослушивателей похож на API перехватчиков, но он не подразумевает блокировку между фазами выполнения компонента. Выдаваемые сигналы асинхронны, поэтому они не должны влиять на общую задержку компонента. Но с распространением состояния следует быть осторожными. Когда вы передаете код компоненту вызывающей стороны, вы не знаете, будет ли он изменен клиентом. Поэтому чрезвычайно важна неизменяемость состояния, распространяемого в API прослушивателей.

Как правило, чем выше гибкость API, тем больше сложности вводится в архитектуру. Это может быть сложность либо кода, либо модели выполнения, если нужно ввести асинхронную обработку. На рис. 4.8 сравниваются эти два подхода: гибкость и сложность.

Если сравнить эти решения, абстрагирование библиотеки метрик обеспечивает некоторую гибкость, но остается явный контракт API, который должен

выполняться клиентами. С другой стороны, API перехватчиков или прослушивателей намного более гибкие, но они раскрывают внутренние события или состояние API, с которым клиент может сделать все что угодно. Этот вариант обеспечивает отличную гибкость. Но при этом нет возможности делать предположения о клиентском коде; необходимо защищаться от непредсказуемых сбоев, для чего используется неизменяемое состояние. Другой недостаток в том, что невозможно делать предположения относительно модели одновременного выполнения клиента (если оно блокирующее или асинхронное). Из-за этого стремление к гибкости системы иногда создает слишком много хлопот. Необходимо найти оптимальную точку на оси «гибкость — сложность» и проектировать систему соответствующим образом.

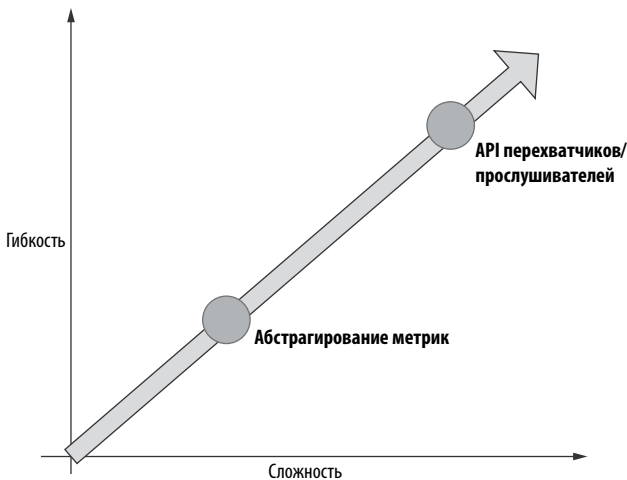


Рис. 4.8. Гибкость и сложность

Мы рассмотрели небольшое подмножество паттернов и способов улучшения гибкости API. Можно использовать и другие паттерны — Декоратор, Фабрика (Factory), Заместитель (Proxy) и т. д. В этой главе мы постарались показать преимущества и недостатки рассмотренных паттернов. Если какое-то решение обеспечивает значительную гибкость, стоит проанализировать присущую ему сложность. Эти общие правила распространяются на все паттерны программирования.

В следующей главе мы покажем, что преждевременная оптимизация — не всегда плохо. Также вы узнаете, в каких случаях оптимизация *критического пути* может быть рациональной, и научитесь находить критические пути в коде, чтобы принимать верные решения об оптимизации его частей.

ИТОГИ

- Для повышения гибкости API можно абстрагировать стороннюю логику.
- Введение API перехватчиков и прослушивателей обеспечивает максимальную гибкость кода. Эти API помогают сделать код расширяемым с точки зрения клиента.
- Стремление к универсальности и гибкости повышает сложность системы.
- Сложность может затрагивать не только код и его обслуживание, но и другие части системы.
- Если вы стремитесь к повышению расширяемости кода за счет использования API перехватчиков, следует тщательно продумать обработку сбоев и сложность модели выполнения, обусловленную этим API.
- Неизменяемость делает поведение системы предсказуемым.
- Использование разных паттернов ведет к различным компромиссам между сложностью и гибкостью.

Преждевременная оптимизация и оптимизация критического пути: решения, влияющие на производительность кода

https://t.me/it_boooks

В этой главе:

- ✓ Когда преждевременная оптимизация — зло.
- ✓ Поиск критического пути в коде с помощью измерений и тестирования производительности.
- ✓ Оптимизация критического пути.

В компьютерной теории существует поговорка: «Преждевременная оптимизация — корень всех зол». И это зачастую справедливо на практике. Не имея данных об объеме ожидаемого трафика и SLA, трудно сделать предположения относительно кода и необходимой производительности. В такой ситуации оптимизацию случайных путей в коде можно сравнить с пальбой вслепую. Вы только усложняете код без разумных на то причин.

ПРИМЕЧАНИЕ

SLA определяет объем трафика, который должен обрабатываться сервисом. В нем также может указываться количество запросов к выполнению, а также число запросов, которые должны обрабатываться с задержкой ниже заданного порогового значения. Аналогично существуют нефункциональные требования (non-functional requirements, NFR), задающие ожидаемую производительность системы.

На стадии проектирования вы уже представляете, с каким объемом трафика должна справляться система. В этой ситуации можно разработать тесты производительности, отражающие трафик в условиях реальной эксплуатации. Если вы смоделируете трафик, то сможете проанализировать пути в коде и найти *критический путь* — фрагмент кода, берущий на себя большую часть работы и выполняющийся почти для каждого пользовательского запроса. В этой главе вы узнаете, что для поиска и оценки критических путей может применяться принцип Парето. Когда критический путь найден, можно заняться его оптимизацией.

Кто-то скажет, что это *преждевременная оптимизация*, поскольку мы оптимизируем код еще до того, как он будет внедрен в эксплуатацию. Однако при наличии достаточных данных можно принять рациональные решения, позволяющие добиться заметных улучшений производительности еще до запуска системы. Данные собираются в результате тестов производительности, выполняемых до развертывания приложения. Определив SLA и ожидания относительно реального трафика в системе, можно моделировать предполагаемый трафик. А после сбора достаточного количества данных, подкрепляющих эксперименты и гипотезы, оптимизация перестает быть преждевременной.

В этой главе основное внимание уделяется поиску критических путей в коде и тестированию их производительности. Вы увидите, как улучшить код с гарантией повышения производительности приложения. Но сначала необходимо понять, когда преждевременная оптимизация действительно является злом или по крайней мере источником проблем.

5.1. КОГДА ПРЕЖДЕВРЕМЕННАЯ ОПТИМИЗАЦИЯ — ЗЛО

Часто мы пишем код приложения, не имея значимых входных данных относительно ожидаемого трафика. В идеальном мире информация о требованиях к ожидаемой пропускной способности и максимальной задержке доступна всегда. На практике часто приходится адаптироваться к конкретной ситуации. Мы начинаем с написания продукта, который просто обслуживать и легко изменять. Однако у нас еще нет жестких требований к производительности. В таком случае попытки оптимизировать код заранее сталкиваются со слишком большим количеством неизвестных.

При оптимизации производительности программного пути мы часто усложняем его. Впрочем, иногда его части приходится писать по конкретной схеме. В таких частях системы производительность достигается за счет сложности. Это может быть сложность кода либо сложность обслуживания или системы используемых компонентов. Без входных данных о трафике может оказаться, что имеющийся код не влияет на общую эффективность основного рабочего процесса. Из-за

этого мы вносим дополнительную сложность, которая не компенсируется повышением производительности.

Другая возможная ловушка — оптимизация кода, основанная на ложных предположениях. Посмотрим, как легко допустить эту ошибку.

5.1.1. Создание конвейера обработки учетных записей

Возьмем простой сценарий: имеется класс, представляющий учетную запись. Требуется построить конвейер обработки, который находит учетную запись с заданным идентификатором. Этот класс представлен в следующем листинге.

Листинг 5.1. Построение сущности, представляющей учетную запись

```
public class Account {
    private String name;
    private Integer id;
    // Конструкторы, методы чтения и записи свойств опущены.
}
```

Код работает со списком учетных записей и получает искомый идентификатор в аргументе. Следующий листинг содержит логику фильтрации учетных записей.

Листинг 5.2. Исходная логика фильтрации

```
public Optional<Account> account(Integer id) {
    return accounts.stream().filter(v -> v.getId().equals(id)).findAny();
}
```

Этот простой код использует API потоков данных; он уже скрывает многие оптимизации производительности. Абстракция потоков работает по ускоренной схеме, а значит, операция фильтрации, проверяющая совпадение идентификатора учетной записи с аргументом, выполняется только в том случае, если эта запись еще не найдена.

МЕТОДЫ FINDANY() И FINDFIRST()

Стоит сказать пару слов о функциях `findAny()` и `findFirst()`, которые часто используются в неправильном контексте. Ускоренная обработка достигается при использовании `findAny()`. Этот метод прекращает обработку при обнаружении любого элемента. `findFirst()` же воспроизводит поведение последовательной обработки. Если обработка разбита на части, `findAny()` может оказаться эффективнее, потому что нас не интересует порядок обработки. С другой стороны, `findFirst()` означает, что обработка должна выполняться последовательно, что замедляет конвейер обработки. Эти различия становятся более важными при использовании параллельных потоков данных.

Данный код был создан как отправная точка для дальнейшей доработки. Важно заметить, что никакой информации о производительности у нас еще нет. Обрабатываемый список может ограничиваться парой элементов, а может содержать миллионы учетных записей. Без этой информации трудно оптимизировать производительность обработки.

Для нескольких учетных записей код будет достаточно хорош. Но для миллионов элементов лучше распределять работу по разным потокам. Одно из возможных решений — создать потоки вручную, разбить работу на пакеты и распределить их по потокам. Также можно воспользоваться существующими механизмами — например, параллельными потоками данных, которые скрывают процесс создания потока и распределяют работу.

Проблема в том, что здесь используются некоторые утверждения о коде, которые могут оказаться ложными. Можно смело предположить, что код обработает максимум N элементов, где N равно 10 000. Пока системный анализ базируется на этих числах, можно заняться оптимизацией соответствующей части кода. К сожалению, таких входных данных часто нет. Оптимизировать код в таком контексте проблематично, потому что это создает дополнительную сложность без очевидных преимуществ. Посмотрим, как ошибочные утверждения усложняют код.

5.1.2. Оптимизация обработки на основании ложных утверждений

Допустим, вы решили ввести оптимизацию производительности в код обработки. Сейчас обработка происходит в одном потоке. Это означает, что работа не распределяется и не выполняется параллельно, чтобы в ней были задействованы все ядра процессора. Одна из возможных оптимизаций — алгоритм *перехвата работы*, для которого работу необходимо разбить на N независимых стадий; все входные данные состоят из N элементов. Схема изображена на рис. 5.1.



Рис. 5.1. Перехват работы как средство оптимизации производительности

Сначала объем работы делится на два потока. На этом этапе каждый поток отвечает за обработку половины от N элементов. Затем код выполняет еще одно разбиение, поскольку задействованы не все потоки, так что работа будет разделена на четверти от N элементов. Теперь каждый поток может начать фактическую обработку. Фаза разбиения должна разбивать элементы на части, соответствующие количеству доступных потоков или ядер. В следующем листинге показано, как компактно записать предлагаемую логику с использованием API потоков данных.

Листинг 5.3. Перехват работы с использованием `parallelStream()`

```
public Optional<Account> accountOptimized(Integer id) {
    return accounts.parallelStream().filter(v ->
        v.getId().equals(id)).findAny();
}
```

Метод `parallelStream()` разбивает работу на N частей. Он использует внутренний пул параллельной обработки (<http://mng.bz/Axro>) с количеством потоков, равным количеству ядер, — 1. Решение выглядит просто, но скрывает значительную сложность. Самое важное изменение заключается в том, что код становится многопоточным, а это означает, что обработка не должна обладать состоянием (например, нельзя изменять состояние в методе обработки, применяемом в качестве фильтра). Так как мы задействуем пул потоков, необходимо контролировать его использование и загруженность.

Другая скрытая сложность, которую создает алгоритм перехвата работы, — фаза разбиения работы. Данный этап занимает дополнительное время и снижает производительность кода. Потери могут превышать выгоду от распараллеливания.

А поскольку работа по оптимизации базируется на ложных утверждениях (или вообще ни на чем), невозможно предсказать, как этот код поведет себя в реальных условиях. Чтобы убедиться в том, что оптимизация производительности эффективна, нужно написать бенчмарк-тест, проверяющий производительность обоих методов.

5.1.3. Оценка оптимизации производительности

Как вы, возможно, помните, обработка применяется к N элементам, где N равно 10 000. Как бы то ни было, если это число основано на эмпирических данных или наших утверждениях, следует как минимум написать тест производительности для проверки результатов оптимизации.

Код тестирования генерирует N случайных учетных записей с идентификаторами от 0 до 10 000. Для этого создается случайная строка с использованием

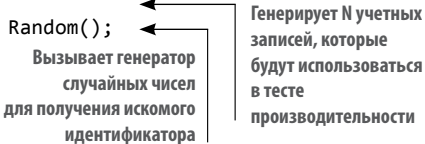
класса `UUID`. Параметр `fork` указывает, что все тесты должны выполняться на одной JVM. Для этого требования воспользуемся программой JMH (Java Microbenchmark Harness) (<https://github.com/openjdk/jmh>). На других платформах имеются инструменты для правильной организации тестирования кода — например, `BenchmarkDotNet` (<https://benchmarkdotnet.org/>) для .NET. Тест производительности полон нюансов; рекомендуем потратить время и изучить проверенные средства для вашей платформы, а не пытаться соорудить собственную реализацию.

До запуска логики тестирования необходимо провести фазу разогрева (`warmup`), которая позволяет JIT оптимизировать кодовые пути. Она настраивается аннотацией `@Warmup`. Мы проведем 10 итераций измерений, чего вполне достаточно — чем больше итераций, тем более повторяемыми будут результаты. Нас интересует среднее время выполнения метода, а результаты выдаются в миллисекундах (мс). Логика инициализации приведена в следующем листинге.

Листинг 5.4. Инициализация теста производительности

```
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Fork;
import org.openjdk.jmh.annotations.Measurement;
import org.openjdk.jmh.annotations.Mode;
import org.openjdk.jmh.annotations.OutputTimeUnit;
import org.openjdk.jmh.annotations.Warmup;
import org.openjdk.jmh.infra.Blackhole;

@Fork(1)
@Warmup(iterations = 1)
@Measurement(iterations = 10)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class AccountsFinderPerformanceBenchmark {
    private static final List<Account> ACCOUNTS =
        IntStream.range(0, 10_000)
            .boxed()
            .map(v -> new Account(UUID.randomUUID().toString(), v))
            .collect(Collectors.toList());
    private static final Random random = new Random();
    // Методы тестирования
```



Вызывает генератор случайных чисел для получения искомого идентификатора

Генерирует N учетных записей, которые будут использоваться в тесте производительности

Метод `baseline()` выполняет первую версию логики поиска учетных записей. Затем метод `parallel()` выполняет улучшенную версию, в которой используется класс `parallelStream`, как показано в следующем листинге.

Листинг 5.5. Реализация логики теста производительности

```

@Benchmark
public void baseline(Blackhole blackhole) {
    Optional<Account> account =
        new AccountFinder(ACCOUNTS)
        ➔ .account(random.nextInt(10_000));
        blackhole.consume(account);
}

@Benchmark
public void parallel(Blackhole blackhole) {
    Optional<Account> account =
        new AccountFinder(ACCOUNTS)
        ➔ .accountOptimized(random.nextInt(10_000))
        blackhole.consume(account);
}

```

Поиск учетной записи со случайным идентификатором

Потребляет результат, чтобы JIT-компилятор знал, что он где-то используется

Логика параллельной версии полностью совпадает

Выполним логику тестирования и посмотрим результаты. На вашем компьютере точные значения могут быть другими, но общая тенденция останется неизменной. В следующем листинге показаны результаты выполнения логики на моем компьютере. Обратите внимание: производительность двух решений почти одинакова.

Листинг 5.6. Просмотр результатов тестирования

```

C\>SN05.premature.AccountsFinderPerformanceBenchmark.baseline
➔ avgt    10    0.027 ± 0.002  ms/op
C\>SN05.premature.AccountsFinderPerformanceBenchmark.parallel
➔ avgt    10    0.030 ± 0.002  ms/op

```

Параллельная обработка может быть чуть медленнее из-за затрат на разбиение, предшествующих реальной работе. Однако при увеличении количества элементов можно заметить, что параллельная версия работает чуть быстрее. В целом различия между двумя решениями пренебрежимо малы.

Из этого простого теста видно, что результаты производительности параллельного решения не добавляют сложности, обусловленной его многопоточностью. Однако сложность кода не возрастает независимо от того, выбран ли `parallelStream` или стандартный поток данных. Сложность скрывается во внутреннем устройстве метода `parallelStream()`. Кроме того, результаты оптимизации могут быть другими в условиях реальной эксплуатации, потому что мы выбрали усовершенствования производительности на основании ложных утверждений.

В таком сценарии преждевременная оптимизация кода (до получения информации о том, как он будет использоваться в реальных условиях) может создать проблемы.

Подведем итог: мы приложили усилия для оптимизации конкретных частей кода системы. Как выяснилось, это не принесло реальной пользы. В сущности, мы только потратили время из-за ложных утверждений. Предполагалось, что код будет выполняться для конкретного количества элементов. В таком контексте вторая версия кода не показала заметных улучшений. Проблема в том, что числа для тестирования были выбраны наугад. Возможно, в реальной системе количество обрабатываемых элементов будет заметно отличаться (в большую или меньшую сторону). Таким образом, у нас будет больше эмпирических данных для оптимизации кода, на этот раз основанных на реальных утверждениях. В этом случае к оптимизации можно будет вернуться позже, но уже с верными данными.

Если вы заранее знаете, что данные учетных записей будут расширяться, необходимо адаптировать код тестирования. При достижении некоторого порога можно заметить, что `parallelStream()` работает лучше, чем стандартный метод `stream()`. В таком случае оптимизация уже не будет преждевременной.

Мы рассмотрели только один из аспектов входных данных, необходимых для эффективной оптимизации производительности. В коде реальных систем существует множество путей. Даже если предположить, что нам известно значение N для обработки всего ввода, оптимизация всех путей может оказаться невыполнимой. Мы должны знать, насколько часто будет выполняться тот или иной путь, чтобы решить, стоит ли тратить время на его оптимизацию. Некоторые пути (например, инициализация) выполняются относительно редко. С другой стороны, существуют пути, выполняемые для каждого пользовательского запроса. Они называются *критическими*. Оптимизация их кода часто эффективна и обеспечивает значительное повышение производительности всей системы. В следующем разделе вы узнаете, как рассуждать при поиске критических путей в коде.

5.2. КРИТИЧЕСКИЕ ПУТИ В КОДЕ

В предыдущем разделе был приведен пример оптимизации, основанной на ложных предположениях. Также вы увидели, что одна из важнейших характеристик данных, полезных при оптимизации кода, — число входящих элементов (N). Это может быть количество запросов в секунду или файлов, которые нужно прочитать. Как вы знаете, сложность алгоритма может вычисляться по количеству входящих элементов (N). Можно не только выбрать наиболее подходящий алгоритм, но и оценить использование памяти.

Знать N очень важно, но в реальных системах не весь код приложения имеет одинаковое значение. Например, рассмотрим простое приложение HTTP с несколькими конечными точками, выполняемыми с разной частотой. Частота запросов представлена на рис. 5.2.

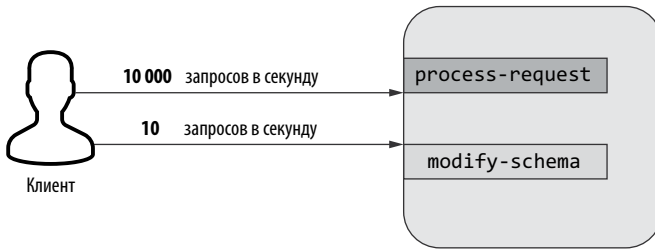


Рис. 5.2. Конечные точки с разными частотами запросов

Первая конечная точка представляет основной функционал приложения. Она обрабатывает почти для каждого вызова со стороны клиента и выполняет основную работу приложения. Допустим, эта точка активизируется клиентами 10 000 раз в секунду. Также можно предположить, что N для обеих конечных точек вычисляется на основании эмпирических данных или SLA, предоставляемых сервисом. В этом примере используемые данные базируются на утверждениях, которые в свою очередь основаны на реальных данных.

С другой стороны, есть метод, выполняющий большую часть «черной работы». Метод `modify-user-details` обрабатывает структуру данных в основной базе данных, используемой приложением HTTP. Он вызывается достаточно *редко*, потому что изменение пользовательских данных — нетипичная задача для клиента. После внесения изменений данные пользователя хранятся в прежней структуре данных в течение долгого времени.

Теперь допустим, что вы измеряете 99-й процентиль задержки для обеих конечных точек (то есть 99 % запросов, обрабатываемых быстрее заданного числа). Через какое-то время вы получаете результаты и заключаете, что задержка p99 точки `process-request` составляет 200 мс, а задержка p99 для `modify-user-details` равна 500 мс. Если рассмотреть эти показатели вне контекста (числа запросов в секунду), можно сделать вывод, что оптимизацию следует начинать с конечной точки `modify-user-details`. Но после добавления контекста о количестве запросов легко заметить, что оптимизация конечной точки `process-request` обеспечит большую экономию ресурсов и времени.

Например, при сокращении задержки p99 для обработки запросов до 20 мс (10 %) достигается общее сокращение задержки в 200 000 мс:

$$(10\,000 \times 200) - (10\,000 \times 180) = 200\,000$$

Но если оптимизировать конечную точку `modify-user-details` вдвое до 250 мс, достигается существенно меньшее общее сокращение задержки в 2500 мс: $(10 \times 500) - (10 \times 250) = 2500$.

На основании этих вычислений можно сделать следующий вывод. Затраты времени на оптимизацию конечной точки, вызываемой чаще, дают в 80 раз большую выгоду, чем в случае точки, оптимизация которой выполняется дольше: $200\,000 \div 2500 = 80$.

Как уже говорилось, путь, выполняемый для большинства запросов, называется *критическим*. Его поиск и оптимизация исключительно важны, если вы хотите оптимизировать производительность каждого приложения.

Оказалось, что в реальных системах трафик достаточно часто неравномерно распределяется между кодовыми путями в приложении. Многие эмпирические исследования показали, что принцип Парето способен упростить анализ систем. Этот принцип рассматривается в следующем разделе.

5.2.1. Принцип Парето в контексте программных систем

В ходе исследования различных систем (организаций, эффективности работы, программных систем) были выявлены присущие большинству из них интересные характеристики. Проанализируем их в контексте программных систем.

Как выяснилось, небольшая доля кода обеспечивает значительную долю ценности программного продукта. На практике чаще всего обнаруживалось соотношение 80 % к 20 %. Другими словами, 80 % ценности и работы, выполняемой системой, производят всего 20 % кода. На рис. 5.3 это соотношение изображено в виде графа.

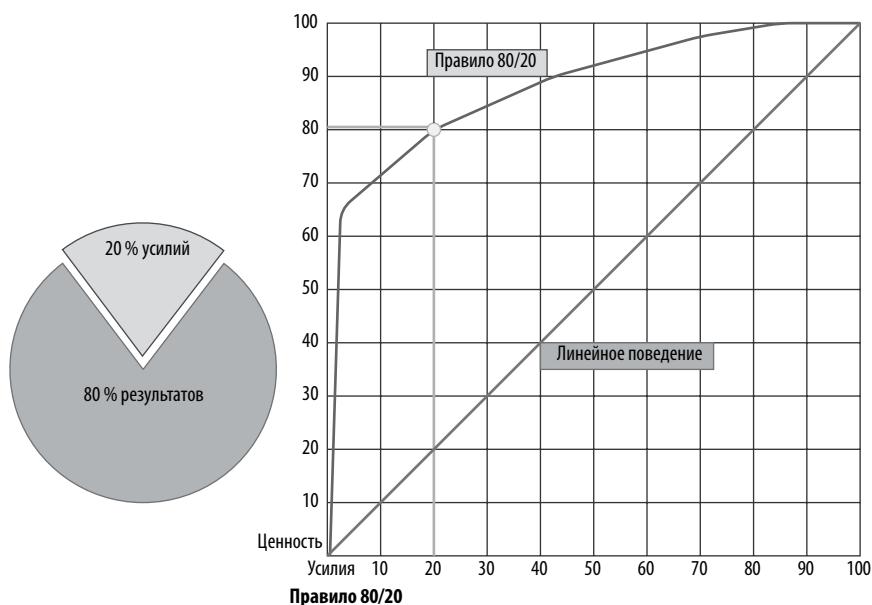


Рис. 5.3. Правило 80/20, представленное принципом Парето

При линейном поведении все пути в коде важны одинаково. В таких сценариях добавление нового компонента в систему означает, что ценность, доставляемая клиенту, пропорционально растет. На практике в каждой системе существует базовая функциональность, несущая наибольшее значение для коммерческих целей. Остальной функционал (проверка данных, обработка граничных случаев и сбоев и т. д.) не критичен, а создаваемая им ценность незначительна (скажем, 20 %). Однако на его реализацию уходит 80 % времени и усилий.

Конечно, реальные пропорции различаются в зависимости от предметной области и системы. Соотношение может быть 30 % к 70 % или даже 10 % к 90 %. Реальное значение роли не играет.

Какой самый важный вывод из этого следует? Оптимизация небольшой части кодовой базы отразится на значительной части клиентов.

При создании новой системы нужны требования SLA с предполагаемой верхней границей трафика, который способна обработать система. Владея этими данными, можно создать тесты производительности, моделирующие реальный трафик.

5.2.2. Настройка количества параллельных пользователей (поток) для заданного уровня SLA

Допустим, сервис должен обеспечивать уровень SLA для обработки 10 000 запросов в секунду. Средняя задержка составляет 50 мс. Если вы хотите рассмотреть такую систему в инструменте анализа производительности, важно задать правильное количество потоков (параллельных пользователей) для выполнения запросов к тестируемой системе.

Если выбрать один поток, можно обрабатывать не более 20 запросов в секунду ($1000 \text{ мс} \div 50 \text{ мс} = 20$). Такая конфигурация не позволит проанализировать SLA системы. Но зная, что один поток обрабатывает 20 запросов в секунду, можно вычислить общее количество необходимых потоков. Ожидаемое количество запросов в секунду делится на число запросов, которые могут обрабатываться одним потоком: $10\,000 \div 20 = 500$.

Результат показывает, что для насыщения системы или сетевого трафика потребуется 500 потоков. Теперь можно настроить инструмент согласно этой характеристике. Если инструмент нагрузочного тестирования не сможет создать столько потоков в одном узле, трафик можно разделить на N узлов тестирования, где каждый узел обрабатывает часть трафика. Например, можно выполнять запросы от четырех узлов. Значит, каждый узел должен выполнять запросы 125 параллельных пользователей ($500 \text{ потоков} \div 4 \text{ узла} = 125$). Вычисления могут немного отличаться в зависимости от инструмента.

Если инструмент использует цикл событий (неблокирующий ввод/вывод), можно выполнять больше запросов из одного потока. В таком случае сначала нужно измерить количество запросов, с которыми может справиться один поток, и адаптировать остальные вычисления к этому числу. Затем следует создать чуть больше потоков, чем показал расчет, поскольку он основан на средней задержке. При этом могут существовать выбросы, замедляющие параллельные потоки. Чтобы посчитать выбросы, можно рассмотреть задержку высоких процентилей (например, p90, p95, p99). Общее количество потоков, необходимое для среднего SLA, умножается на некий коэффициент (скажем, 1,5), чтобы создать дополнительные потоки на случай временного замедления тестируемой системы.

Наконец, можно измерить критические пути для некоторого количества активизаций и определить затраченное время. С этими числовыми данными можно обнаружить критический путь и вычислить, насколько значительную выгоду мы получим от оптимизации небольшого фрагмента кода. Благодаря характеристикам многих систем, следующим принципу Парето в отношении оптимизации критического пути, можно внести усовершенствования, влияющие на большинство клиентов. В следующем разделе мы применим эту схему для оптимизации системы с определенным уровнем SLA: построим новую систему с предметной областью. Используя новые знания, мы оптимизируем ее критический путь.

5.3. СЛОВАРНЫЙ СЕРВИС С ПОТЕНЦИАЛЬНЫМ КРИТИЧЕСКИМ ПУТЕМ

Допустим, вы хотите построить словарный сервис, который предоставляет две функциональности по двум конечным точкам API. На рис. 5.4 изображена архитектура сервиса.

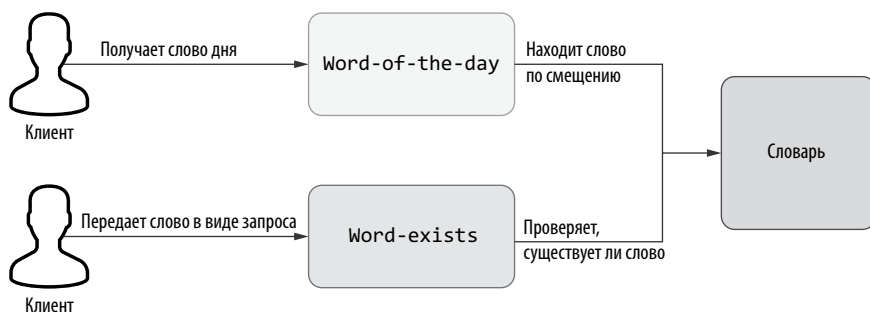


Рис. 5.4. Архитектура словарного сервиса с двумя видами функциональности

Первая предоставляемая функциональность — получение «слова дня» (word-of-the-day). Система вычисляет смещение, соответствующее текущей дате, и возвращает слово с индексом, равным смещению.

Вторая функциональность проверяет, что слово существует (word-exists). Пользователь передает слово в параметре запроса, а сервис ищет его в словаре, возвращая в теле ответа информацию о его существовании. В следующем листинге представлен словарный сервис, являющийся базовым компонентом системы; в его основе лежит интерфейс `WordsService`.

Листинг 5.7. Реализация интерфейса `WordsService`

```
public interface WordsService {
    String getWordOfTheDay();
    boolean wordExists(String word);
}
```

Метод `getWordOfTheDay()` не принимает аргументы. Он просто возвращает запрошенное слово. Метод `wordExists()` получает проверяемое слово и возвращает признак, сообщающий, есть оно в словаре или нет. Первая реализация `WordsService` не пытается применять преждевременные оптимизации, так как у нас еще нет числовых данных, относящихся к SLA или трафику.

5.3.1. Получение «слова дня»

Базовая функциональность для получения «слова дня» вычисляет индекс для заданной даты. В следующем листинге представлена простая логика вычисления — в ней используется год и день года, а также коэффициент для лучшего распределения возвращаемых слов.

Листинг 5.8. Получение «слова дня»

```
private static final int MULTIPLY_FACTOR = 100;
private static int getIndexForToday() {
    LocalDate now = LocalDate.now();
    return now.getYear() + now.getDayOfYear() * MULTIPLY_FACTOR;
}
```

ПРИМЕЧАНИЕ

Мы выбрали множитель 100, но это может быть любое произвольное число.

Реализация словарного сервиса должна получать в аргументе путь к фактическому файлу словаря, чтобы загрузить файл и сканировать его содержимое. Функцию вычисления индекса сегодняшнего дня можно макетировать передачей функции-поставщика, как показано в следующем листинге. Это может пригодиться для модульного тестирования, потому что тест не должен базироваться на состоянии, возвращаемом вызовом `LocalDate.now()`.

Листинг 5.9. Добавление конструктора DefaultWordsService

```

public class DefaultWordsService implements WordsService {
    private static final int MULTIPLY_FACTOR = 100;
    private static final IntSupplier DEFAULT_INDEX_PROVIDER =
        DefaultWordsService::getIndexForToday;
    private Path filePath;
    private IntSupplier indexProvider;

    public DefaultWordsService(Path filePath) {
        this(filePath, DEFAULT_INDEX_PROVIDER);
    }

    @VisibleForTesting
    public DefaultWordsService(Path filePath,
        IntSupplier indexProvider) {
        this.filePath = filePath;
        this.indexProvider = indexProvider;
    }

```

DefaultWordsService реализует WordsService

Поставщик вызывает функцию, которая использует локальную дату

В аргументе должен передаваться путь к словарию

Второй конструктор нужен только для модульного тестирования

Использует значение Int, полученное от поставщика, как индекс «слова дня»

Логика вычисления «слова дня» использует класс `Scanner` (<http://mng.bz/ZzjR>), который позволяет выполнить отложенное сканирование файла. Если вам нужна следующая строка, вызовите метод, который ее получает. Когда обработка будет завершена, загружать дополнительные строки уже не нужно.

Логика примера весьма проста. Она перебирает файл, пока не будет найден индекс, обозначающий «слово дня». Если остались строки, логика продолжает выполняться. Если текущий обрабатываемый индекс равен индексу искомого слова, мы возвращаем слово и завершаем обработку. Логика получения «слова дня» приведена в следующем листинге.

Листинг 5.10. Добавление метода `getWordOfTheDay()`

```

@Override
public String getWordOfTheDay() {
    int index = indexProvider.getAsInt();
    try (Scanner scanner = new Scanner(filePath.toFile())) {
        int i = 0;
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            if (index == i) {
                return line;
            }
            i++;
        }
    } catch (FileNotFoundException e) {
        throw new RuntimeException("Ошибка в getWordOfTheDay для индекса: " +
            filePath, e);
    }
    return "Сегодня слово отсутствует.";
}

```

Получает индекс текущего дня

Передает сканеру путь к файлу словаря

Получает следующую строку в виде String

Обратите внимание: в конце обрабатывается один граничный случай. Если индекс «слова дня» был слишком большим, слово не возвращается. Это может произойти при выходе индекса за границы используемого файла словаря.

5.3.2. Проверка существования слова

Вторая функциональность, которую предоставляет сервис, — проверка присутствия конкретного слова в словаре. Логика получения этой информации напоминает логику «слова дня», но, чтобы проверить существование слова, необходимо перебрать весь файл. Метод `wordExists()` ищет слово, переданное в аргументе. Если строка, загруженная из файла, равна аргументу `word`, возвращается `true`; это значит, что слово существует в словаре. Наконец, если слово не найдено после перебора всего файла, возвращается `false`. Функциональность проверки представлена в следующем листинге.

Листинг 5.11. Добавление метода `wordExists()`

```
@Override
public boolean wordExists(String word) {
    try (Scanner scanner = new Scanner(filePath.toFile())) {
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            if (word.equals(line)) {
                return true;
            }
        }
    } catch (FileNotFoundException e) {
        throw new RuntimeException("Ошибка в wordExists для слова: " + word, e);
    }
    return false;
}
```

Логика `wordExists()` не оптимизирована, потому что мы не определили условия SLA. Тестов для определения производительности текущего решения еще нет, но теперь можно предоставить доступ к логике через конечную точку API.

5.3.3. Предоставление доступа к `WordsService` с использованием сервиса HTTP

Класс `WordsController` предоставляет две конечные точки, как видно из листинга 5.12. Первая точка, `/word-of-the-day`, использует GET-запрос HTTP, который не получает параметров. Запрос запускает загрузку файла в словарь, а после загружает файл `words.txt` из папки `resources`. Функциональность первой конечной точки предоставляется по пути API `/word-of-the-day API`. (Все пути в этом примере используют префикс `/words`.) Вторая функциональность предоставляется через конечную точку `/word-exists`. Она использует слово, переданное в параметре запроса, и проверяет, существует ли оно.

Листинг 5.12. Добавление класса WordsController

```

@Path("/words")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class WordsController {
    private final WordsService wordsService;

    public WordsController() {
        java.nio.file.Path defaultPath =
            Paths.get(
                Objects.requireNonNull(
                    getClass().getClassLoader().getResource("words.txt")).getPath());
        wordsService = new DefaultWordsService(defaultPath);
    }
    @GET
    @Path("/word-of-the-day")
    public Response getAllAccounts() {
        return Response.ok(wordsService.getWordOfTheDay()).build();
    }
    @GET
    @Path("/word-exists")
    public Response validateAccount(@QueryParam("word") String word) {
        boolean exists = wordsService.wordExists(word);
        return Response.ok(String.valueOf(exists)).build();
    }
}

```

Конструирует реализацию WordsService по умолчанию

Упаковывает «слово дня» в тело ответа HTTP

Упаковывает информацию о существовании слова в ответе HTTP

Наконец, приложение HTTP запускается с использованием встроенного сервера HTTP Dropwizard (<http://mng.bz/REpZ>). Приложение должно расширять класс `io.dropwizard.Application`, предоставляющий функциональность запуска сервера HTTP, как показано в листинге 5.13. Из-за этого класс `Application` необходимо расширить с типом по умолчанию `Configuration`. Таким образом создается экземпляр `WordsController`, предоставляющий бизнес-функциональность, который затем регистрируется как конечная точка API. Наконец, приложение запускает веб-сервер HTTP, доступный по адресу `http://localhost:8080/words`.

Листинг 5.13. Запуск сервера HTTP

```

public class HttpApplication extends Application<Configuration> {

    @Override
    public void run(Configuration configuration, Environment environment) {
        WordsController wordsController = new WordsController();
        environment.jersey().register(wordsController);
    }

    public static void main(String[] args) throws Exception {
        new HttpApplication().run("server");
    }
}

```

ПРИМЕЧАНИЕ

Если запустить эту функцию `main`, приложение Words с обоими контроллерами будет работать на локальном компьютере.

В следующем разделе мы применим информацию об ожидаемом трафике, чтобы обнаружить критический путь. Для этого воспользуемся бенчмарк-тестами Gatling (<https://gatling.io/open-source/>) для моделирования трафика и классом Dropwizard MetricsRegistry, чтобы измерить хронометраж кодовых путей. Посмотрим, подчиняется ли структура приложения принципу Парето, описанному в предыдущем разделе.

5.4. ОБНАРУЖЕНИЕ КРИТИЧЕСКОГО ПУТИ В КОДЕ

Допустим, наши оценки трафика и SLA показывают, что конечная точка `word-of-the-day` обслуживает один запрос в секунду. С другой стороны, конечная точка `word-exists` будет вызываться чаще 20 запросов в секунду. Прямолинейные вычисления показывают, что это превышает соотношение принципа Парето (правило 80/20):

$$1 \div (20 + 1) = \sim 5 \%$$

$$20 \div (20 + 1) = \sim 95 \%$$

Формула показывает, что функциональность `word-exists` обслуживает 95 % запросов пользователя и не обслуживает только 5 %. Но прежде чем оптимизировать эту конечную точку, следует создать тест производительности для обеих точек, чтобы получить данные о задержках. Зная количество запросов и величину задержек, можно вычислить общую выгоду от оптимизации той или иной функциональности. Воспользуемся инструментом Gatling для тестирования производительности.

5.4.1. Применение Gatling для создания тестов производительности API

Мы хотим смоделировать два сценария тестирования производительности. Первый предназначен для конечных точек `word-of-the-day` и выполняет один запрос в секунду. Продолжительность измерений составляет 1 минуту для быстрого получения обратной связи. Этого достаточно для нашего случая, в котором сравниваются исходная и оптимизированная версии. При тестировании реальных систем измерения длятся намного дольше.

Модели с использованием Gatling пишутся на языке Scala, при этом каждая модель должна расширять класс `Simulation`. Сценарий получения «слова дня» прямолинеен. Требуется выполнить GET-запрос для заданной конечной точки, и каждый запрос будет выполняться в контексте URL `http://localhost:8080/words`. Если вы захотите развернуть приложение Words на отдельном сервере, этот URL нужно будет изменить. Конечная точка API получает и генерирует формат JSON. Сценарий теста выполняет GET-запрос HTTP для конечной точки `/word-of-the-day`. Ожидается, что результатом будет код ответа HTTP 200. Любой другой код рассматривается как ошибка. Реализация приведена в следующем листинге.

Листинг 5.14. Проверка производительности word-of-the-day

```
class WordsSimulation extends Simulation {
  val httpProtocol = http
    .baseUrl("http://localhost:8080/words")
    .acceptHeader("application/json")

  val wordOfTheDayScenario = scenario("word-of-the-day") ← Этот сценарий
    .exec(WordOfTheDay.get)                               используется для
                                                          генерирования трафика

  object WordOfTheDay {
    val get = http("word-of-the-day").get("/word-of-the-day").check(status is
      200)
  }
}
```

Второй сценарий похож на первый, но GET-запрос HTTP должен отправить проверяемое слово как параметр HTTP, поэтому сценарию необходимо передать список проверяемых слов. В следующем листинге приведены слова файла из примера `words.csv`.

Листинг 5.15. Слова, используемые для тестирования производительности

```
word
1Abc
bigger
presence
234
zoo
```

В список входит слово **bigger**, находящееся в начале словаря. Также имеется слово **presence** из середины. Наконец, слово **zoo** находится в конце словаря. Также имеются два несуществующих слова, для которых отработает полное сканирование файла.

Сценарий проверки использует файл `words.csv` и передает его в параметре запроса конечной точке API. **feeder** получает слова из `words.csv` и использует их случайным образом. Наконец, сценарий выполняет GET-запрос с параметром `word`. Код этого сценария приведен в следующем листинге.

Листинг 5.16. Проверка производительности word-exists

```

val validateScenario = scenario("word-exists") ← Сценарий word-exists выполняет
    .exec(ValidateWord.validate)                логику проверки

object ValidateWord {
    val feeder = csv("words.csv").random
    val validate = feed(feeder).exec(
        http("word-exists")
            .get("/word-exists?word=${word}").check(status is 200)
    )
}

```

Когда сценарии будут заданы, их следует внедрить в исполнительное ядро и указать ожидаемый трафик. В листинге 5.17 приведен пример, как это сделать. Первый сценарий обрабатывает один запрос в секунду. Второй сценарий (проверка), отвечающий за 95 % клиентских запросов, выполняет 20 запросов в секунду.

Листинг 5.17. Настройка профиля трафика

```

setUp(
    wordOfTheDayScenario.inject(
        constantUsersPerSec(1) during (1 minutes)
    ),
    validateScenario.inject(
        constantUsersPerSec(20) during (1 minutes)
    ).protocols(httpProtocol)
)

```

Теперь можно приступить к тестированию. Сначала необходимо запустить `HttpApplication`. Когда приложение заработает на локальном хосте, можно начать тесты Gatling с помощью команды `mvn gatling:test`. Она запускает тестирование производительности приложения. Через некоторое время результаты будут доступны в виде веб-страницы в формате HTML.

Проанализируем результаты производительности для обоих сценариев. Как видно на рис. 5.5, результаты для сценария `word-of-the-day` довольно неплохие.

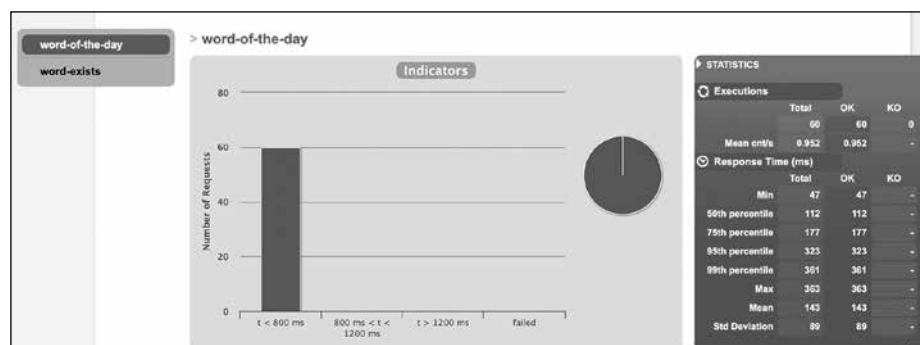


Рис. 5.5. Просмотр исходных результатов производительности word-of-the-day

Все запросы к конечной точке `/word-of-the-day` успешно завершаются быстрее 800 мс. Задержка p99 равна 361 мс.

Перейдем к результатам сценария проверки слов. Как показано на рис. 5.6, эта конечная точка выполнила большинство запросов.

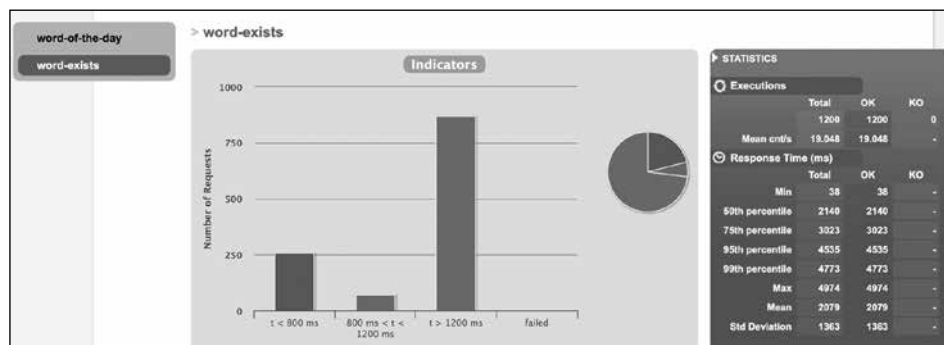


Рис. 5.6. Просмотр исходных результатов производительности word-exists

Большинство запросов к конечной точке `/words` имеет задержку, превышающую 1200 мс. Здесь значение p99 составляет почти 5 с.

Сравнивая результаты, видим проблемы с производительностью `word-exists`. Ее улучшение затронет 95 % клиентов. Преждевременно оптимизировать `word-of-the-day` не нужно, так как производительность этой конечной точки достаточно хороша и затрагивает всего 5 % клиентов.

Чтобы вычислить влияние обеих конечных точек на производительность, воспользуемся формулой из второй главы. У `word-of-the-day` задержка p99 составляет 360 мс, но выполняется всего один запрос в секунду: $(1 \times 360) = 360$. С другой стороны, у `word-exists` задержка p99 составляет почти 5000 мс: $(20 \times 5000) = 100\,000$. Можно вычислить, что `word-of-the-day` отвечает менее чем за 1 % работы по обработке запросов: $360 / (100\,000 + 360) = 0,003 = 0,3 \%$.

После этих вычислений становится очевидно, где сосредоточить усилия по оптимизации. Логика `word-exists` занимает 99,7 % общей рабочей нагрузки системы.

Когда вы знаете, что логика `word-exists` создает проблемы, необходимо получить информацию с нижнего уровня кода. Нужно понимать, на какие части кодового пути приходится большая часть времени обработки. Чтобы это понять, можно измерить хронометраж кода на критическом пути; займемся этим в следующем разделе.

5.4.2. Измерение хронометража кодовых путей с использованием MetricRegistry

В разделе 5.3 код, проверяющий существование слова в словаре, был простым, и в нем не применялась оптимизация. На тот момент мы еще не знали, нужна ли она. Теперь у нас есть входные данные о количестве запросов, которые будет обрабатывать сервис.

Тесты производительности показали, что проблема с задержкой возникает с конечной точкой `/word-exists`, обслуживающей 95 % пользовательских запросов.

Тесты Gatling работали по принципу «черного ящика»; это означает, что у нас была информация о поведении конкретных конечных точек, но мы не знали, на какие части системы приходятся основные затраты времени. Посмотрим, как обстоит дело сейчас.

Метод `wordExists()` включает два основных аспекта функциональности. Первый загружает файл с проверяемыми словами. Второй — фаза сканирования — определяет, существует ли фактическое слово. Обе стадии можно упаковать в отдельные таймеры, чтобы измерить продолжительность каждого вызова этих методов и получить более подробную информацию об их производительности. В листинге 5.18 создаются два таймера. Первый таймер измеряет время, необходимое для загрузки файла. Второй таймер измеряет время сканирования (то есть время, необходимое для проверки того, есть ли слово в словаре).

Листинг 5.18. Измерение хронометража логики `word-exists`

```
@Override
public boolean wordExists(String word) {
    Timer loadFile = metricRegistry.timer("loadFile");
    try (Scanner scanner = loadFile.time(() -> new
        Scanner(filePathToFile())) {
        ← Измеряет время создания нового
           сканера для обращения к файлу

    Timer scan = metricRegistry.timer("scan");
    return scan.time(
        () -> {
            ← Измеряет время выполнения
               основной логики метода
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                if (word.equals(line)) {
                    return true;
                }
            }
            return false;
        });
    } catch (Exception e) {
        throw new RuntimeException("Ошибка в wordExists для слова: " + word, e);
    }
}
```

Таймер выполняется для каждой операции. В выходных данных будут получены проценты, среднее значение и количество вызовов. Измерение можно проводить на любом уровне детализации, соответствующем актуальным потребностям. Измерения всех кодовых путей могут повлиять на общую производительность логики обработки, так что проводить их следует с осторожностью. Когда логика будет оптимизирована, можно принять решение об удалении некоторых (или всех) измерений.

Последний шаг перед повторным выполнением тестов производительности — использование нового класса `MeasuredDefaultWordsService` в `WordsController`. Код его создания приведен в следующем листинге.

Листинг 5.19. Использование `MeasuredDefaultWordsService`

```
wordsService = new MeasuredDefaultWordsService(defaultPath);
```

При перезапуске приложения будет измеряться каждый запрос к конечной точке API `/word-exists`. После завершения теста производительности Gatling можно обратиться к конечной точке `http://localhost:8081/metrics?pretty=true`, чтобы увидеть все метрики, предоставляемые приложением. Найдите раздел, посвященный `loadFile`; в нем приводятся данные процентов. Для нас наиболее интересен 99-й процентиль; см. следующий листинг.

Листинг 5.20. Просмотр данных о производительности `loadFile`

```
loadFile": {  
  "count": 1200,  
  "p99": 0.000730684,  
  "duration_units": "seconds"  
}
```

Результаты выводятся в секундах, и видим, что 99-й процентиль действия загрузки файла равен 7 мс. Операция загрузки файла не вызывает проблем с производительностью, обнаруженных с помощью тестов Gatling.

Счетчик `count` содержит количество вызовов конкретного кода. По этим данным можно сравнить разные кодовые пути и определить, где тратится большая часть времени. Такая возможность полезна, когда заранее определенная информация об ожидаемом трафике или SLA отсутствует. Если эти сведения доступны, можно воспользоваться метриками для проверки утверждений. В таком сценарии мы развертываем приложение в рабочей конфигурации с метриками и вычисляем, какие кодовые пути активизируются большую часть времени. Узнав это, можно найти критический путь и сосредоточиться на повышении его производительности. В следующем листинге приведен хронометраж сканирования файла.

Листинг 5.21. Хронометраж сканирования

```
"scan": {
  "count": 1200,
  "p99": 4.860273076,
  "duration_units": "seconds"
}
```

Видим, что 99-й перцентиль составляет почти 5 с. Похоже, мы нашли причину проблем с производительностью. Операция сканирования происходит долго, и именно на нее приходится большая часть времени обработки запроса.

После выявления причины можно переходить к оптимизации критического пути. Сделаем это в следующем разделе и определим, привело ли усовершенствование к повышению производительности.

ПРИМЕЧАНИЕ

Если добавить код хронометража в приложение, производительность которого тестируется, нельзя, попробуйте использовать профилирование, чтобы получить более подробную информацию о времени, затрачиваемом в конкретных частях кода. Например, в JVM можно воспользоваться Java Flight Recorder (<http://mng.bz/2jYg>).

5.5. ПОВЫШЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ КРИТИЧЕСКОГО ПУТИ

Мы хотим оптимизировать кодовый путь `word-exists`. Когда вы экспериментируете с методом `wordExists()` и пытаетесь применить другой подход, необходимо получить обратную связь о его производительности. Можно воспользоваться существующими тестами Gatling, но они работают на высоком уровне, а их выполнение занимает больше времени. Для этого необходимо запустить веб-сервер и тесты Gatling и получить результаты. Так как мы знаем конкретный кодовый путь, который необходимо оптимизировать, можно написать низкоуровневые микротесты для этого пути. Это позволит быстрее получить обратную связь и найти более производительное решение.

Стоит заметить, что при наличии высокоуровневых тестов производительности создавать микротесты для каждого изменения, скорее всего, не обязательно. Микротесты требуют усилий, но, с другой стороны, обеспечивают более быструю обратную связь. Если вы хотите протестировать N решений для одной низкоуровневой проблемы, микротесты могут оказаться более эффективными.

Далее я покажу, как реализовать микротесты для учебных целей. Впрочем, вы можете разработать свое решение или написать другой микротест и сравнить его с представленным в этом разделе.

5.5.1. Создание микротеста JMН для существующего решения

Прежде чем оптимизировать кодовый путь, напомним тест JMН для существующего кода. Будем считать его эталонным и использовать как отправную точку для дальнейшего улучшения кода. Тест покрывает логику критического пути, который занимает большую часть времени обработки запросов.

В листинге 5.22 приведена логика подготовки бенчмарк-теста. Он выполняет 10 итераций для тестового прогона (чем больше итераций, тем точнее результаты). Требуется измерить среднее время, занимаемое методом теста. Один тест проводит измерения вызова `wordExists` `NUMBER_OF_CHECKS * WORDS_TO_CHECK.size()` раз. Каждая итерация выполняет 100 проверок для моделирования более реалистичного сценария использования. Словарный сервис будет использоваться 100 раз, после чего начнется следующая итерация.

Листинг 5.22. Создание теста `word-exists`

```
@Fork(1)
@Warmup(iterations = 1)
@Measurement(iterations = 10)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class WordExistsPerformanceBenchmark {
    private static final int NUMBER_OF_CHECKS = 100;
    private static final List<String> WORDS_TO_CHECK =
        Arrays.asList("made", "ask", "find", "zones", "task", "123");
```

Обратите внимание: мы выбираем слова из начала, середины и конца словарного файла. Также в список включены некоторые несуществующие слова.

Эталонный тест создает `DefaultWordsService` (текущая логика без оптимизации). Проверка существования слова будет выполнена 100 раз, и каждое слово из списка будет проверено один раз за итерацию.

Экземпляр `WordsService` создается один раз для каждой итерации измерений JMН. Он повторно используется `100 * WORDS_TO_CHECK.size()`. Метод `wordExists()` вызывается для каждого слова, как показано в следующем листинге.

Листинг 5.23. Получение эталонных данных

```
@Benchmark
public void baseline(Blackhole blackhole) {
    WordsService defaultWordsService = new DefaultWordsService(getWordsPath());
    for (int i = 0; i < NUMBER_OF_CHECKS; i++) {
        for (String word : WORDS_TO_CHECK) {
            blackhole.consume(defaultWordsService.wordExists(word));
```

```

    }
  }
}

private Path getWordsPath() {
    try {
        return Paths.get(
            Objects.requireNonNull(getClass().getClassLoader()
                .getResource("words.txt")).toURI());
    } catch (URISyntaxException e) {
        throw new IllegalStateException("Invalid words.txt path", e);
    }
}

```

Получает путь к файлу словаря

Benchmark	Mode	Cnt	Score	Error	Units
CH05.WordExistsPerformanceBenchmark.baseline	avgt		55440.923		ms/op

После получения эталонных данных можно попытаться создать оптимизированный вариант метода `wordExists()` и добавить бенчмарк. Так мы проверим, повлияла ли оптимизация на производительность. Эталонные результаты сообщают количество миллисекунд на операцию. На их основе можно сделать вывод о том, как улучшенная версия соотносится с базовой.

5.5.2. Оптимизация проверки с использованием кэширования

Будем считать, что файл со словами для проверки существования слов статичен и не изменяется. Это утверждение важно в контексте нашей логики. Оно означает, что результат однократной проверки существования слова не будет меняться в будущем.

Можно построить статическую карту, в которой ключом является слово, а значением — его существование в словаре. Строить ее необходимо на стадии инициализации приложения. На рис. 5.7 изображена теоретическая карта для рассматриваемого примера.

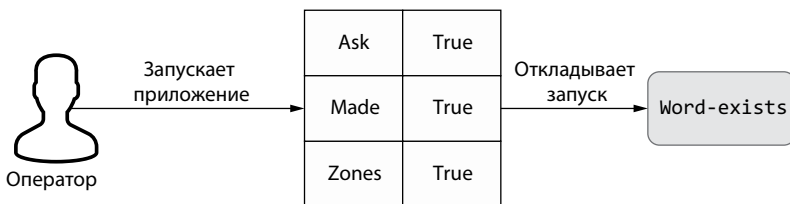


Рис. 5.7. Немедленная инициализация и вычисление

Файл словаря может содержать миллионы записей, и построение карты с опережением существенно увеличивает время запуска приложения; в данном случае применяется немедленная оптимизация. Это также означает, что нам придется использовать значительные ресурсы для предварительного вычисления данных, которые могут не понадобиться в будущем.

Некоторая часть оперативной памяти будет задействована независимо от фактического использования сервиса. Может оказаться, что проверяется лишь небольшая часть слов, а другие остаются невостребованными. Ненужные слова занимают место без всякой пользы, а программа использует больше памяти, чем необходимо.

Другое возможное решение — отложенное конструирование кэша. Это означает, что кэш изначально пуст и строится при поступлении запросов. Мы предполагаем, что словарный файл статичен и не изменяется. Если он содержит небольшой объем данных, его можно кэшировать в течение неопределенного времени без вытеснения. Тем не менее в реально используемых системах, которые должны загружать гораздо больший объем данных, ситуация может быть другой. Например, приложение, которое проверяет существование слов в большем количестве языков (например, английском, испанском, китайском и т. д.), должно загружать все словари. В таких случаях избыточные затраты памяти необходимо сократить, поэтому можно воспользоваться вытеснением данных, которые находятся в кэше в течение некоторого времени.

Время вытеснения может вычисляться на основании данных трафика. Например, регистрация запросов в журнале может дать статистику по запрашиваемым словам. На основании времени запросов можно определить интервалы между ними. Следующий шаг — построение статического распределения временных интервалов. Имея такие данные, например, можно определить 90-й перцентиль и задать время вытеснения для данного значения. Это гарантирует, что кэш будет обслуживать 90 % запросов. А если 99-й перцентиль будет не слишком большим, то для вытеснения также можно выбрать это значение.

Наше решение предназначено для ситуации, в которой приложение еще не было развернуто, и у нас нет достаточно данных о распределении трафика, кроме SLA и ожидаемого количества запросов в секунду. В таком случае можно выбрать прогнозное значение и сохранить статистику по содержимому кэша. Когда приложение будет выпущено в эксплуатацию, можно собрать информацию о попаданиях в кэш и промахах, а также другую статистику, чтобы оценить эффективность работы кэша. Если доля промахов слишком велика, стоит подумать об увеличении времени вытеснения.

Реализуем решение на базе кэширования и измерим его производительность (листинг 5.24). Необходимо сконструировать кэш, который вызывает существующий метод `word-exists`, если заданное слово отсутствует в кэше. Для этого

устанавливается время вытеснения по умолчанию (5 минут). При поступлении дополнительных данных о распределении трафика в рабочей среде это значение адаптируется. В кэше ключом будет слово, а значением — его наличие в словаре. Для этой цели воспользуемся классом `LoadingCache` из Guava (<http://mng.bz/1jOX>).

ПРИМЕЧАНИЕ

Мы используем библиотеку Google Guava, потому что это одна из самых популярных библиотек кэширования для Java. Можно выбрать и другую библиотеку кэширования (например, Caffeinate) — общие выводы этой главы останутся справедливыми.

Если к конкретному слову не было обращений за период вытеснения, оно удаляется из кэша. Следующий листинг показывает, как конструируется кэш.

Листинг 5.24. Конструирование кэша для `word-exists`

```
public static final Duration DEFAULT_EVICTION_TIME = Duration.ofMinutes(5)

LoadingCache<String, Boolean> wordExistsCache =
    CacheBuilder.newBuilder()
        .ticker(ticker)
        .expireAfterAccess(DEFAULT_EVICTION_TIME)
        .recordStats()
        .build(
            new CacheLoader<String, Boolean>() {
                @Override
                public Boolean load(@Nullable String word) throws Exception {
                    if (word == null)
                        return false;
                    return checkIfWordExists(word);
                }
            }
        );
```

Сохранение статистики для получения информации об эффективности кэширования

Если word содержит null, немедленно возвращается false

В противном случае выполняется реальный метод проверки

Прежде чем тестировать производительность улучшенного решения, проверим его на правильность. Метод `FakeTicker()` будет использоваться для моделирования течения времени без потока приостановки, как показано в листинге 5.25. Первая проверка существования слова запускает реальную операцию проверки. После нее в кэше должен появиться один элемент.

Листинг 5.25. Модульное тестирование кэша `word-exists`

```
@Test
public void shouldEvictContentAfterAccess() {
    // Дано
    FakeTicker ticker = new FakeTicker();
    Path path = getWordsPath();
    CachedWordsService wordsService = new CachedWordsService(path, ticker);

    // Если
```

```

assertThat(wordsService.wordExists("make")).isTrue();

// To
assertThat(wordsService.wordExistsCache.size()).isEqualTo(1);
assertThat(wordsService.wordExistsCache.stats()
➔ .missCount()).isEqualTo(1);
assertThat(wordsService.wordExistsCache.stats()
➔ .evictionCount()).isEqualTo(0);

// Если
ticker.advance(
➔ CachedWordsService.DEFAULT_EVICTION_TIME);
assertThat(wordsService
➔ .wordExists("make")).isTrue();

// To
assertThat(wordsService.wordExistsCache.stats()
➔ .evictionCount()).isEqualTo(1);
}

```

Первый запрос иницирует реальную загрузку данных

Элемент не был вытеснен; он находится в кэше

Продвижение времени для моделирования вытеснения

Вызывает wordExists() для активизации вытеснения

После операции элемент вытесняется: счетчик вытеснений = 1

Наконец, в листинге 5.26 показано, как написать микротест с помощью JMH для проверки того, улучшает ли новая архитектура производительность `wordExists()`. Единственное отличие в том, что мы используем реализацию, основанную на кэше.

Листинг 5.26. Написание микротеста для кэша word-exists

```

@Benchmark
public void cache(Blackhole blackhole) {
    WordsService defaultWordsService = new CachedWordsService(getWordsPath());
    for (int i = 0; i < NUMBER_OF_CHECKS; i++) {
        for (String word : WORDS_TO_CHECK) {
            blackhole.consume(defaultWordsService.wordExists(word));
        }
    }
}

```

Повторим тест и сравним результаты первой эталонной версии и улучшенной версии с кэшем. Результаты представлены в следующем листинге.

Листинг 5.27. Результаты бенчмарк-теста для эталонной версии и версии с кэшем

Benchmark	Mode	Cnt	Score	Error	Units
CH05.WordExistsPerformanceBenchmark.baseline	avgt		55440.923		ms/op
CH05.WordExistsPerformanceBenchmark.cache	avgt		557.029		ms/op

Можно заключить, что средняя производительность решения увеличилась в 100 раз. Это превосходный результат, и можно перейти к сквозному тестированию всего приложения. Чтобы приложение Words использовало новую реализацию на базе кэша, необходимо внести только одно изменение: инициализировать ее в `WordsService`. В следующем листинге показано, как это делается.

Листинг 5.28. Использование CacheWordsService в WordsController

```
wordsService = new CachedWordsService(defaultPath);
```

Все готово к выполнению тестов производительности Gatling. Чтобы провести тестирование, выполните действия, описанные в разделе 5.4. Откройте результаты теста производительности (рис. 5.8).

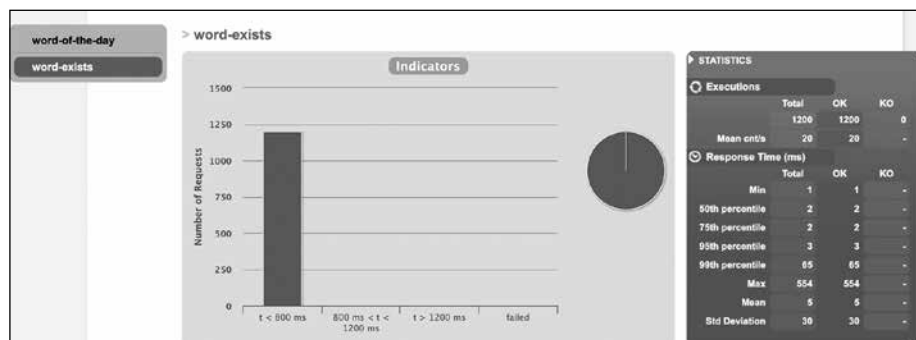


Рис. 5.8. Тесты производительности для улучшенной версии word-exists

Видим, что производительность решения существенно возросла. Задержка 99-го перцентиля составляет 65 мс — это почти в 80 раз быстрее, чем в начальной версии!

После оптимизации следует пересчитать объем работы этих конкретных частей кода во время выполнения. Задержка p99 сократилась до 65 мс. Для вычисления эффекта логики word-of-the-day и word-exists можно воспользоваться формулой из раздела 5.2:

- Трафик word-exists составляет 20 запросов в секунду, задержка p99 равна 65 мс:

$$(20 \times 65) = 1300$$

- Конечная точка word-of-the-day обрабатывает один запрос в секунду, задержка p99 равна ~360 мс:

$$(1 \times 360) = 360$$

Наконец, можно вычислить процент трафика, генерируемый обеими конечными точками. Для этого вычислим трафик word-of-the-day:

$$360 \div (360 + 1300) \approx 0,21 = 21 \%$$

Как видно из рис. 5.9, вычисления показывают, что трафик word-of-the-day генерирует 21 % рабочей нагрузки системы, а трафик word-exists отвечает за

остальные 79 %. Нагрузка **word-exists** сократилась с 99,7 %, хотя она все еще отвечает за бóльшую часть работы. Тем не менее, как следует из расчетов выше, это влияет на 95 % запросов пользователей. После оптимизации функциональность получения «слова дня» (затрагивающая 5 % запросов пользователей) получает 21 % вычислительных ресурсов. При необходимости дальнейшей оптимизации можно рассчитать возможную экономию времени по формулам из раздела 5.2. Допустим, производительность обеих конечных точек можно улучшить еще на 10 %. Для «слова дня» эти формулы дают 36 мс экономии времени, потому что эта конечная точка получает всего один запрос в секунду: $0,1 \times 360 \times 1 = 36$.

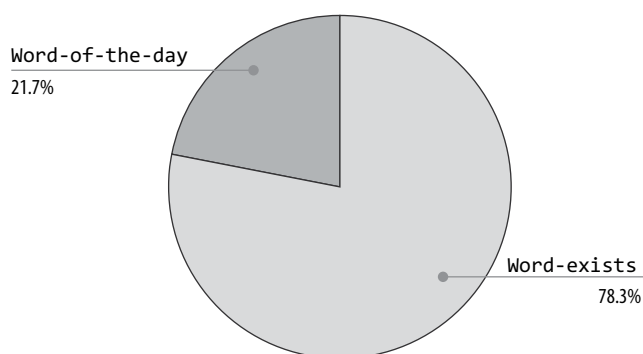


Рис. 5.9. Процентное соотношение трафика **word-of-the-day** и **word-exists**

Улучшение производительности **word-exists** на дополнительные 10 % дает экономию 130 мс, так как обрабатываются 20 запросов в секунду: $0,1 \times 65 \times 20 = 130$.

Однако может оказаться, что оптимизация производительности **word-exists** еще на 10 % непрактична или труднодостижима. Можно вычислить это только путем оптимизации **word-of-the-day** на 40 %, что сэкономит нам больше времени, чем оптимизация **word-exists** на 10 %: $0,4 \times 360 \times 1 = 144$.

Если оптимизация этой конечной точки на 40 % предпочтительнее, чем оптимизация **word-exists** на 10 %, можно выбрать вариант оптимизации фрагментов кода, не находящихся на критическом пути. Но как вы заметили, оптимизация критического пути дает больше преимуществ при меньших усилиях. В реальных условиях проще и эффективнее оптимизировать кодовый путь на 10 %, а не на 40 %.

ПРИМЕЧАНИЕ

При оптимизации производительности одной конечной точки часто требуется выделить больше ресурсов для обработки трафика. При этом снижаются задержки и/или увеличивается пропускная способность конкретной конечной точки, но может

оказаться, что эти ресурсы приходится перенаправлять с других точек. В результате может пострадать их производительность. По этой причине важно контролировать производительность всех конечных точек, доступных для клиентов, и следить, чтобы производительность других точек не сокращалась. Кроме того, чем больше ресурсов используется, тем выше нагрузка на узел, на котором работает приложение. В какой-то момент приложение придется масштабировать на большее количество узлов (горизонтальное масштабирование). Другой вариант — увеличение мощности узлов (вертикальное масштабирование). Таким образом, повышение производительности часто (но не всегда) означает необходимость выделения дополнительных ресурсов, потребность в которых влияет на масштабируемость.

5.5.3. Увеличение количества входящих слов в тестах производительности

Осталось сделать еще одно важное замечание. В окончательной версии решения логика построена на основе кэширования. Значит, тесты всего шести входящих слов недостаточно эффективны для проверки производительности. Решение, в основе которого лежит механизм кэширования, следует тестировать с большим количеством входящих слов. Это позволяет удостовериться, что данные из кэша не вытесняются слишком рано. Кроме того, такой подход гарантирует, что кэш не расходует чрезмерный объем памяти в системе.

Возьмем 100 случайных слов из словаря и поместим их в файл `words.csv`, используемый симуляцией Gatling. Количество слов должно соответствовать ожидаемому трафику. Наш тест выполняет 20 запросов в секунду в течение 60 с, итого 1200 запросов. Если использовать еще более случайные слова (например, 1000), почти каждый запрос будет попадать в кэш с не загруженным ранее значением и наблюдаемого прежде улучшения производительности не будет.

Можно делать иначе: выбирать более случайные слова с увеличением времени тестирования. Тем самым мы заполним кэш данными. Тогда последующие запросы будут чаще приходиться на элементы, которые уже существуют в кэше.

Чтобы получить N случайных слов из файла `words.txt`, можно воспользоваться командой Linux `sort`. В следующем листинге приведен код получения случайных слов.

Листинг 5.29. Получение случайных слов

```
sort -R words.txt | head -n 100 > to_check.txt
```

Наконец, необходимо скопировать слова из файла `to_check.txt` в файл `words.csv`, используемый Gatling, и снова запустить симуляцию. Как видно из рис. 5.10, результаты заметно отличаются.

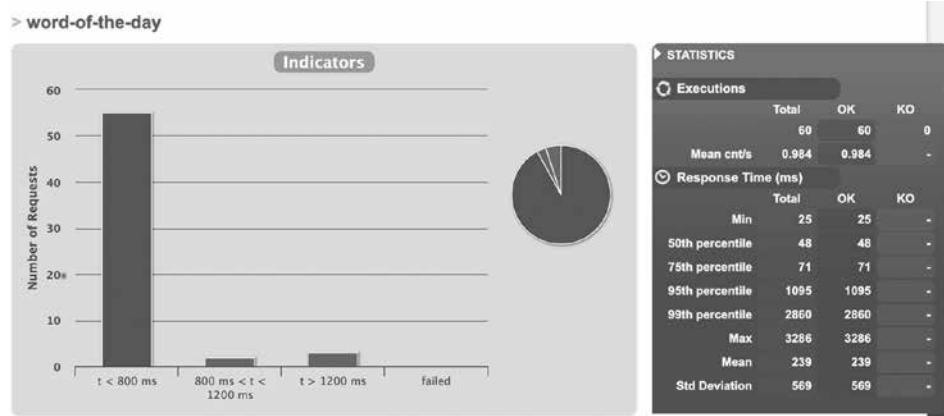


Рис. 5.10. Увеличение количества входящих слов

Производительность все еще намного выше, чем у исходного решения, но и задержки увеличились. Это объясняется тем, что почти 10 % ($100 \div 1200$) запросов приходится на пустой кэш.

Экспериментируйте с разными паттернами трафика и временем тестирования. Главное, что необходимо усвоить из этого раздела: при изменении деталей решения (например, параметров кэша) стоит адаптировать к ним тесты производительности, чтобы представить более реалистичное распределение трафика.

Если вы проводите анализ и собираете данные перед выпуском продукта, то такую оптимизацию уже нельзя назвать преждевременной. Бенчмарк-тесты дают много полезной информации о коде. Если у вас есть тесты с трафиком, близким к реальному, вы получите правдоподобные данные о производительности. Имея такие сведения, можно оптимизировать нужные части кода и быть уверенными, что это даст хорошие результаты.

В этой главе нам удалось сократить задержку и повысить производительность приложения до его развертывания. Если у вас достаточно данных об SLA и ожидаемом объеме трафика, преждевременная оптимизация может дать отличные результаты. Главное — не забывать следовать стратегии поиска узких мест и не увлекаться оптимизацией случайных кодовых путей. Если ваше приложение реализует функциональность и следует принципу Парето в отношении распределения трафика в коде, найти критический путь относительно просто. А когда он будет найден, область оптимизации можно сократить при помощи микротестов. Они позволят повысить эффективность оптимизации кода за счет более быстрого цикла получения обратной связи. Не забудьте, что при оптимизации критического пути узкое место может переместиться в другие части кода.

В следующей главе вы узнаете, как простота пользовательского интерфейса может повысить затраты на обслуживание. Также мы рассмотрим некоторые преимущества абстрагирования используемых систем и сопутствующие компромиссы.

ИТОГИ

- Даже если код не на критическом пути, его выполнение может занимать много времени. Это часто происходит, когда некритический кодовый путь на порядок медленнее критического; такой код может требовать оптимизации.
- Для поиска критических путей, на которых необходима оптимизация, можно воспользоваться формулами из второго раздела этой главы.
- Бенчмарк-тесты, созданные с применением инструмента Gatling, позволяют обнаружить критический путь на основании ожидаемого трафика.
- Отдельные части кода можно измерять с использованием метрик.
- Чтобы сузить область действия тестов производительности, можно воспользоваться микротестами и JMH.
- Кэширование часто позволяет оптимизировать критический путь.
- Данные тестов Gatling можно использовать для проверки и сравнения результатов оптимизации производительности.

Простота и затраты на обслуживание API

https://t.me/it_boooks

В этой главе:

- ✓ UX и компромиссы обслуживания при интеграции со сторонними библиотеками.
- ✓ Эволюция настроек, предоставляемых клиентам.
- ✓ Плюсы и минусы абстрагирования кода, который вам не принадлежит.

При построении систем для конечных пользователей на первый план выходит простота API и удобный опыт взаимодействия (UX, User Experience). Важно заметить, что понятие UX применимо ко всем интерфейсам. Можно спроектировать графический интерфейс (GUI), элегантный и удобный для пользователя. Также можно создать REST API в UX-ориентированном стиле. На более глубоком уровне даже средства командной строки могут быть (или не быть) UX-ориентированными. По сути, любой программный продукт, взаимодействующий с пользователем, требует обсуждения и планирования в области UX.

Механизм конфигурации системы — точка входа, которую необходимо предоставить клиентам, и жизненно важная часть UX-ориентированности компонента. Часто системы зависят от нескольких компонентов и используют их для получения результата обработки. Каждый из зависимых компонентов предоставляет свои настройки конфигурации, которые необходимо каким-то образом задавать.

Все последующие компоненты (элементы, используемые системой, для которой создается UX) можно абстрагировать и не предоставлять их настройки

непосредственно в инструменте, взаимодействующем с пользователем. Такой подход улучшит UX нашей системы, но потребует значительных затрат на обслуживание.

Противоположный подход — напрямую предоставить доступ ко всем зависимым настройкам. Такое решение не потребует значительных усилий по обслуживанию, но у него есть свои недостатки. Во-первых, клиенты жестко привязываются к компонентам, используемым сервисом или программой. Это усложняет возможную модификацию компонента. Кроме того, вносить UX-ориентированные изменения в такой компонент (как и те, что подразумевают обратную совместимость) станет сложно или вовсе невозможно.

Оба варианта имеют свои плюсы и минусы, и с позиции этих решений можно обсуждать связи между простотой API и затратами на обслуживание. Далее в этой главе мы рассмотрим эти компромиссы подробнее, а начнем с представления и анализа облачного компонента, в котором уже есть собственный механизм конфигурации. Затем мы используем этот компонент в двух инструментах с разным соотношением затрат на обслуживание и простоты UX.

6.1. БАЗОВАЯ БИБЛИОТЕКА, ИСПОЛЬЗУЕМАЯ ДРУГИМИ ИНСТРУМЕНТАМИ

Чтобы система создавала ценность для бизнеса, она должна интегрироваться с другими программными инструментами и использовать их в своей работе. Например, приложение может интегрироваться с базами данных, очередями или провайдерами облачных сервисов. Возможно, понадобится интеграция с такими частями операционной системы, как файловая система, сетевые интерфейсы или диск. Большинство систем, от которых зависят приложения, предоставляют собственные пакеты SDK (Software Development Kit) или клиентские библиотеки. Они позволяют легко интегрироваться с системами без разработки полной интеграции с нуля. Рисунок 6.1 иллюстрирует сказанное.

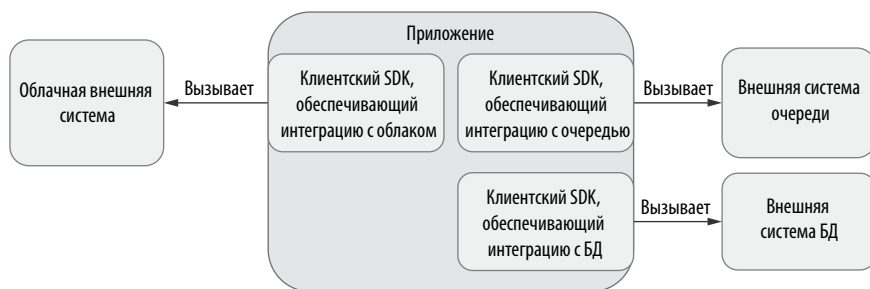


Рис. 6.1. SDK с клиентом, обеспечивающим интеграцию с внешними системами

Как уже было отмечено, у реального приложения может возникнуть необходимость в интеграции с базами данных, очередями (например, Apache Kafka или Pulsar) и облачными сервисами (например, EC2 и GCP). Все эти сервисы предоставляют клиентские библиотеки. Использование этих библиотек позволяет создать более надежный продукт по сравнению с тем, в котором каждую интеграцию приходится писать с нуля.

Почти каждый клиент нуждается в настройке конфигурации, прежде чем взаимодействовать со сторонними системами. Например, это могут быть учетные данные аутентификации, продолжительность тайм-аута, размер буфера или другие настройки. Конфигурация может предоставляться, например, через системные свойства, переменные окружения или файлы конфигурации, и все пользователи клиентского компонента должны передать свой вариант конфигурации клиентской библиотеке.

Для примеров этой главы ради ясности и доступности объяснения построим простой компонент облачного клиента, который нужно настроить перед использованием. Далее задействуем этот компонент из двух программ, с которыми работают конечные пользователи. Начнем с создания компонента, а также изучим его применение.

6.1.1. Создание клиента облачного сервиса

Наш клиент облачного сервиса даст возможность вызывающей стороне компонента выполнить запрос, загружающий данные в облачный сервис. Перед выполнением запроса выполняется аутентификация. Существуют два основных способа аутентификации запросов. Первый использует для проверки маркер безопасности, а второй — имя пользователя и пароль. Стратегия аутентификации выбирается в зависимости от конфигурации, заданной пользователем. Две альтернативы изображены на рис. 6.2.

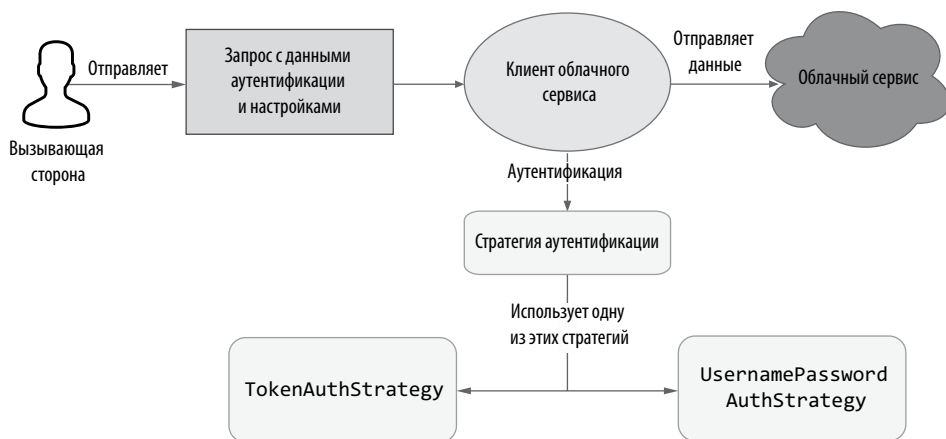


Рис. 6.2. Компонент облачного клиента и две возможные стратегии аутентификации

Рассмотрим эти компоненты. Первая точка входа облачного компонента для вызывающей стороны — класс `Request`. Он содержит список элементов данных и информацию, необходимую для выполнения аутентификации: имя пользователя, пароль или маркер. В следующем листинге представлена эта реализация (если аннотация `Nullable` вам незнакома, см. <http://mng.bz/PWgw>).

Листинг 6.1. Создание облачного запроса

```
public class Request {
    @Nullable private final String token;
    @Nullable private final String username;
    @Nullable private final String password;
    private final List<String> data;
    // Конструкторы, hashCode, равенство,
    // методы чтения и записи не приводятся
}
```

← Сообщает пользователям, что маркер может принимать значение null

Компонент `CloudServiceClient` обрабатывает запрос (можно рассматривать компонент как облачную клиентскую библиотеку с облачным провайдером — AWS, Azure, GCP и т. д.). Интерфейс компонента достаточно прост, как показано в следующем листинге. Он предоставляет всего один открытый метод, который должен использоваться клиентами этого компонента.

Листинг 6.2. Создание интерфейса `CloudServiceClient`

```
public interface CloudServiceClient {
    void loadData(Request request);
}
```

Метод `loadData()` получает запрос и загружает его в облачный сервис. Реализация этого метода также выполняет аутентификацию. Рассмотрим некоторые стратегии аутентификации этого компонента, которые могут использоваться клиентом.

6.1.2. Стратегии аутентификации

Наш облачный компонент поддерживает две стратегии аутентификации. Первая — простая аутентификация с именем пользователя и паролем, как показано в листинге 6.3. Она требует, чтобы во входящем запросе имя пользователя и пароль были отличны от `null`. Объект создается на базе этой конфигурации и проверяет, соответствует ли имя пользователя/пароль в `Request` заданной конфигурации.

Листинг 6.3. Стратегия аутентификации с именем пользователя/паролем

```
public interface AuthStrategy {
    boolean authenticate(Request request);
}
```

← `AuthStrategy` реализуется обеими стратегиями аутентификации

```

public class UsernamePasswordAuthStrategy implements AuthStrategy {
    private final String username;
    private final String password;

    public UsernamePasswordAuthStrategy
    ➔ (String username, String password) {
        this.username = username;
        this.password = password;
    }

    @Override
    public boolean authenticate(Request request) {
        if (request.getUsername() == null
    ➔ || request.getPassword() == null) {
            return false;
        }

        return request.getUsername().equals(username) &&
        request.getPassword().equals(password);
    }
}

```

Создает стратегию для имени пользователя и пароля, прочитанных из файла конфигурации

Если имя пользователя или пароль в запросе содержит null, возвращается false

Проверяет совпадение имени пользователя и пароля

Обе стратегии реализуют интерфейс `AuthStrategy`. Если запрос содержит имя пользователя или пароль с неопределенным значением `null`, то метод `authenticate()` возвращает `false`. Заметим, что хранение пароля в формате `String` может создать проблемы из-за риска утечки данных из приложения (и их возможного похищения злоумышленниками). Более эффективный способ хранения будет рассмотрен ниже.

Вторая стратегия аутентификации похожа на первую, но использует маркер безопасности для проверки запроса. Если маркер совпадает со значением, предоставленным конструктором, то `authenticate()` возвращает `true`.

Листинг 6.4. Маркер `AuthStrategy`

```

public class TokenAuthStrategy implements AuthStrategy {
    public TokenAuthStrategy(String token) {
        this.token = token;
    }

    private final String token;

    @Override
    public boolean authenticate(Request request) {
        if (request.getToken() == null) {
            return false;
        }
        return request.getToken().equals(token);
    }
}

```

Реализует `AuthStrategy`

Проверяет совпадение маркеров

Логика не изменилась по сравнению с механизмом аутентификации в листинге 6.4, но в ней используется маркер из запроса.

6.1.3. Понимание механизма конфигурации

Клиент предоставляет конфигурацию облачного сервиса в файле формата YAML.

ПРИМЕЧАНИЕ

Многие реальные фреймворки и библиотеки используют очень похожий механизм конфигурации на базе YAML (например, Spring Boot), и примеры этой главы можно сопоставить с некоторыми из них. Но чтобы не привязывать главу к конкретным механизмам, напомним отдельный код вместо использования готовых решений.

На основании конфигурации в этом файле создадим класс `CloudServiceConfiguration`, используемый реализацией облачного сервиса. На этой стадии построения клиентской библиотеки конфигурация содержит только класс `AuthStrategy`, который будет использоваться в механизме аутентификации. В следующем листинге приведен код создания конфигурации облачного сервиса.

Листинг 6.5. Реализация `CloudServiceConfiguration`

```
public class CloudServiceConfiguration {
    private final AuthStrategy authStrategy;

    public CloudServiceConfiguration(AuthStrategy authStrategy) {
        this.authStrategy = authStrategy;
    }

    public AuthStrategy getAuthStrategy() {
        return authStrategy;
    }
}
```

Загрузка конфигурации из YAML должна быть абстрагирована от реализации `CloudServiceClient`. Такая абстракция может пригодиться, если вы решите поддерживать другой вариант файлов конфигурации — JSON, Hоsom и т. д. Для этого создадим класс `DefaultCloudServiceClient`, внедряющий `CloudServiceConfiguration` через конструктор. Метод `loadData()` сначала проверяет, требует ли запрос аутентификации. В нем используются данные `CloudServiceConfiguration#authStrategy`, предоставленные в объекте конфигурации, как показано в следующем листинге.

Листинг 6.6. Создание `CloudServiceClient` по умолчанию

```
public class DefaultCloudServiceClient implements CloudServiceClient {
    private CloudServiceConfiguration cloudServiceConfiguration;
    public DefaultCloudServiceClient(CloudServiceConfiguration
```

```

        cloudServiceConfiguration) {
            this.cloudServiceConfiguration = cloudServiceConfiguration;
        }

        @Override
        public void loadData(Request request) {
            if (cloudServiceConfiguration.getAuthStrategy().authenticate(request)) {
                insertData(request.getData()); ← После проверки вставляет
            }                                     данные в облачный сервис
        }

```

Остается последний шаг — прочитать файл конфигурации YAML и сконструировать облачный клиент. Файл конфигурации YAML должен содержать раздел с конфигурацией аутентификации. В следующем листинге показано, как файл конфигурации должен определять стратегию с именем пользователя/паролем.

Листинг 6.7. Конфигурация облачного сервиса для имени пользователя/пароля

```

auth:
  strategy: username-password
  username: user
  password: pass

```

Используем значение `strategy (username-password)` для создания правильной реализации `AuthStrategy`. При использовании стратегии с маркером конфигурация YAML выглядит немного иначе. Для целей тестирования в качестве значения маркера можно использовать любой идентификатор UUID. В реальной системе маркеры не будут жестко фиксироваться в коде. Они генерируются динамически и обновляются с некоторым временным интервалом. Стратегия с маркером безопасности представлена в следующем листинге.

Листинг 6.8. Маркер для облачного сервиса

```

auth:
  strategy: token
  token: c8933754-30a0-11eb-adc1-0242ac120002

```

Наконец, сконструируем класс-строитель, отвечающий за загрузку конфигурации и конструирование `DefaultCloudServiceClient`. Используем `ObjectMapper` (<http://mng.bz/J14o>) для чтения файла конфигурации YAML и его разбора. Так как мы используем YAML, файл конфигурации имеет структуру, которую можно описать как «карту карт». Первая внешняя карта включает все настройки, необходимые для секции аутентификации. Вторая содержит другие настройки. На рис. 6.3 изображена высокоуровневая структура «карты карт».

На иллюстрации ключом внутренней карты является имя свойства (например, `strategy`), а значением может быть любой объект (например, `token`). Другая внешняя карта имеет профильный раздел в конфигурации. Раздел `auth`

представляет профильную внешнюю карту. Если позже вы захотите добавить раздел конфигурации, для него будет создан новый профильный раздел (например, `other setting section`).

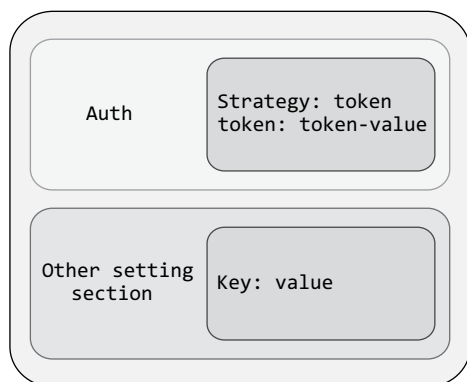


Рис. 6.3. Структура конфигурации в файле YAML

На рис. 6.9 конструктор `CloudServiceClientBuilder` создает карту карт для чтения файла YAML. Также имеются константы с идентификаторами стратегии (например, `USERNAME_PASSWORD_STRATEGY`) для создания правильной стратегии аутентификации. В нашем примере используется класс `YAMLFactory` из библиотеки Jackson (<http://mng.bz/wnGO>). `ObjectMapper` читает конфигурацию из файла YAML.

Листинг 6.9. Конструктор `CloudServiceClientBuilder`

```

public class CloudServiceClientBuilder {
    private static final String USERNAME_PASSWORD_STRATEGY = "username-password";
    private static final String TOKEN_STRATEGY = "token";
    private final ObjectMapper mapper;
    private final MapType yamlConfigType;

    public CloudServiceClientBuilder() {
        mapper = new ObjectMapper(new YAMLFactory());
        MapType mapType =
            mapper.getTypeFactory().constructMapType(HashMap.class, String.class,
Object.class);
        yamlConfigType =
            mapper
                .getTypeFactory()
                .constructMapType(
                    HashMap.class, mapper.getTypeFactory()
                        .constructType(String.class), mapType);
    }
    // ...

```

Используется для чтения файла конфигурации

Так как файл содержит YAML, используется `YAMLFactory`

Внутренняя карта имеет строковый ключ и объектное значение

Внешняя карта содержит тип внутренней карты

Осталось рассмотреть последнюю часть клиента облачного сервиса. Как видно из листинга 6.10, за создание объекта отвечают два метода. Мы разрешаем

вызывающей стороне этого кода передать путь к файлу конфигурации YAML. При этом мы предоставляем возможность обеспечить `CloudServiceConfiguration` на программном уровне без использования механизма конфигурации YAML. Сам факт предоставления двух механизмов конфигурации позволяет вызывающим сторонам настроить клиент двумя способами. У обоих вариантов есть свои достоинства и недостатки, которые мы разберем ниже.

Листинг 6.10. Создание `DefaultCloudServiceClient` на основе конфигурации

```
public DefaultCloudServiceClient
➔ create(CloudServiceConfiguration cloudServiceConfiguration) {
    return new DefaultCloudServiceClient(cloudServiceConfiguration);
}

public DefaultCloudServiceClient create(Path configFilePath) {
    try {

        Map<String, Map<String, Object>> config =
            mapper
                ➔ .readValue(configFilePath.toFile(), yamlConfigType);
        AuthStrategy authStrategy = null;
        Map<String, Object> authConfig = config.get("auth");

        if (authConfig.get("strategy")
            ➔ .equals(USERNAME_PASSWORD_STRATEGY)) {
            authStrategy =
                new UsernamePasswordAuthStrategy(
                    (String) authConfig.get("username"),
                    ➔ (String) authConfig.get("password"));
        } else if (authConfig.get("strategy")
            ➔ .equals(TOKEN_STRATEGY)) {
            authStrategy = new TokenAuthStrategy((String) authConfig.get("token"));
        }
        return new DefaultCloudServiceClient(new
            CloudServiceConfiguration(authStrategy));
    } catch (IOException e) {
        throw new UncheckedIOException("Ошибка при загрузке файла из: " +
            configFilePath, e);
    }
}
```

Передает конфигурацию в конструктор

Читает файл YAML с использованием `configFilePath`

Извлекает раздел конфигурации аутентификации

Если используется стратегия `USERNAME_PASSWORD_STRATEGY`

... создается `UsernamePasswordAuthStrategy`

Аналогичная логика применяется для `TOKEN_STRATEGY`

Метод `create()` на базе YAML читает из конфигурации раздел аутентификации. Затем он проверяет, соответствует ли стратегия `CloudServiceConfiguration`. Если проверка дает положительный результат, логика `create()` пытается сконструировать класс `UsernamePasswordAuthStrategy`. В противном случае, если используется стратегия `TOKEN_STRATEGY`, создается `TokenAuthStrategy`.

Когда библиотека облачного клиента будет готова, можно внедрить два инструмента, которые будут ее использовать. В них применяются разные подходы к интеграции. В первом случае настройки облачного клиента предоставляются

напрямую; вы увидите, как это влияет на стоимость обслуживания. Во втором случае эти настройки абстрагируются, а предоставляются их конфигурации, которые отображаются на облачный сервис. Сначала обсудим вариант с прямым предоставлением настроек.

6.2. ПРЯМОЕ ПРЕДОСТАВЛЕНИЕ НАСТРОЕК ЗАВИСИМОЙ БИБЛИОТЕКИ

Начнем с инструмента, использующего облачный клиент как пакетный сервис. Его главная функция — построение пакета входящих запросов до того момента, когда размер буфера превысит параметр размера пакета. Когда буфер заполняется, программа вызывает облачный клиент, который выполняет аутентификацию и передает данные в облачный сервис (рис. 6.4).

Перед работой клиент должен настроить пакетный сервис. Этот сервис использует облачный клиент, в который также необходимо передать конфигурацию клиента. Она должна быть предоставлена конечным пользователем для построения объекта `CloudServiceClient`, используемого `BatchService`.

Конфигурация пакетного сервиса достаточно проста, так как содержит всего один параметр (размер пакета). Она приведена в листинге 6.11.

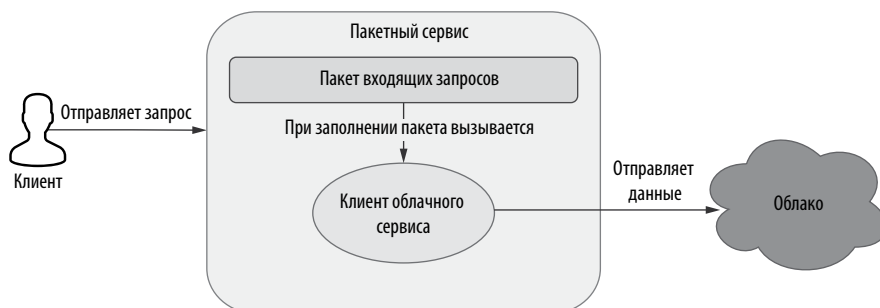


Рис. 6.4. Архитектура пакетного сервиса для облачного клиента

Листинг 6.11. Реализация `BatchServiceConfiguration`

```

public class BatchServiceConfiguration {
    public final int batchSize;

    public BatchServiceConfiguration(int batchSize) {
        this.batchSize = batchSize;
    }

    public int getBatchSize() {
        return batchSize;
    }
}
  
```

Пакетный сервис использует свою конфигурацию для ограничения количества агрегированных событий. Когда пакет содержит достаточно элементов (их количество равно параметру `batchSize` или превышает его), облачный клиент используется для отправки данных. `BatchService` работает с классом `Request`, представленным в предыдущем разделе. Вся логика `BatchService` приведена в следующем листинге.

Листинг 6.12. Логика `BatchService`

```
public class BatchService {
    private final BatchServiceConfiguration batchServiceConfiguration;
    private final CloudServiceClient cloudServiceClient;
    private final List<String> batch = new ArrayList<>();

    public BatchService(
        BatchServiceConfiguration batchServiceConfiguration, CloudServiceClient
        cloudServiceClient) {
        this.batchServiceConfiguration = batchServiceConfiguration;
        this.cloudServiceClient = cloudServiceClient;
    }

    public void loadDataWithBatch(Request request) {
        batch.addAll(request.getData());
        if (batch.size() >=
            batchServiceConfiguration.getBatchSize()) {
            cloudServiceClient.loadData(withBatchData(request));
        }

        private Request withBatchData(Request request) {
            return new Request(request.getToken(), request.getUsername(),
            request.getPassword(), batch);
        }
    }
}
```

Буферизует данные в списке

Использует внедренный `CloudServiceClient`

Когда размер пакета достигает порога или превышает его...

... используется облачный сервис и происходит загрузка данных

Важно, что при выполнении этих запросов пакетный сервис использует облачный клиент напрямую. Внедрение `CloudServiceClient` в конструктор является точкой интеграции между пакетным инструментом и облачным клиентом.

ИНКАПСУЛЯЦИЯ `CLOUDSERVICECLIENT`

В этом примере мы работаем с интерфейсом `CloudServiceClient`, обобщенным в контексте конкретной облачной библиотеки. Если нужно повысить гибкость, можно подумать о создании отдельного класса, инкапсулирующего конкретную разновидность `CloudServiceClient`. Это упростит переключение используемых библиотек без воздействия на код вызывающей стороны (потому что этот код будет использовать облачный клиент через слой абстракции).

6.2.1. Конфигурация пакетного инструмента

Самое важное решение, касающееся UX и обслуживания пакетного сервиса, — способ передачи настроек в используемый облачный клиент. Мы решили, что конечный пользователь пакетного инструмента должен предоставить конфигурацию в файле YAML. Раздел `auth` этого файла передается напрямую в загрузчик конфигурации облачного клиента, как показано на рис. 6.5.

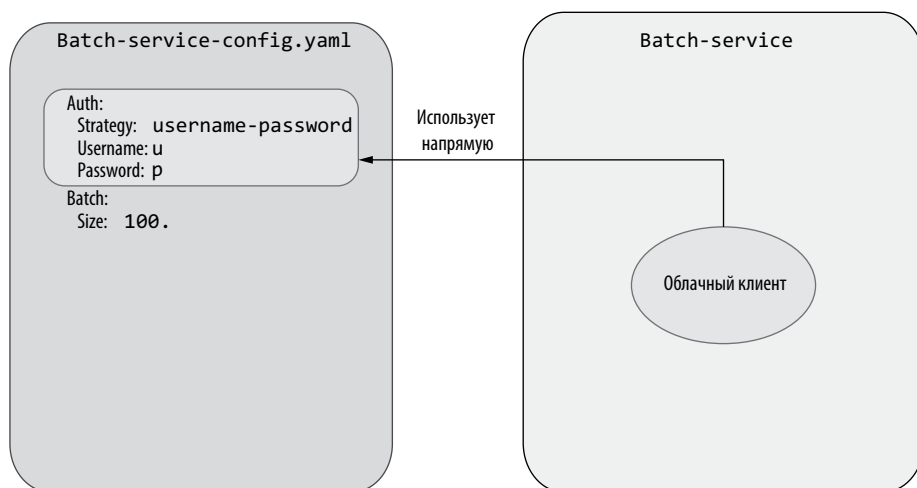


Рис. 6.5. Прямая передача настроек облачного клиента из файла конфигурации YAML

Важно: раздел `auth` конфигурации пакетного сервиса должен иметь такую же структуру, как YAML-разметка, настроенная в облаке. Это также означает, что клиентам пакетного сервиса предоставляется доступ к внутренним подробностям конфигурации облачного клиента. Вследствие этого обслуживание с нашей стороны при конструировании облачной конфигурации не требуется. Клиент пакетного сервиса предоставляет конфигурацию, а пакетный сервис передает ее как есть.

Пакетный сервис использует раздел `batch` конфигурации. Строитель пакетного сервиса получает файл YAML в аргументе и загружает файл. Затем он извлекает раздел `batch` и использует его для конструирования `BatchServiceConfiguration`. Наконец, весь файл YAML передается в строителя клиента облачного сервиса. Как вы, возможно, помните из предыдущего раздела, `CloudServiceClientBuilder` извлекает раздел `auth` из файла и конструирует клиент. Процесс показан в следующем листинге.

Листинг 6.13. Передача файла YAML в облачный клиент

```

public class BatchServiceBuilder {
    public BatchService create(Path configFilePath) {
        try {

            Map<String, Map<String, Object>> config =
                mapper.readValue(configFilePath.toFile(), yamlConfigType);
            Map<String, Object> batchConfig = config.get("batch");
            BatchServiceConfiguration batchServiceConfiguration =
                new BatchServiceConfiguration
                    ➔ ((Integer) batchConfig.get("size"));

            CloudServiceClient cloudServiceClient =
                new CloudServiceClientBuilder()
                    ➔ .create(configFilePath);
            return new BatchService(batchServiceConfiguration, cloudServiceClient);
        } catch (IOException e) {
            throw new UncheckedIOException("Ошибка при загрузке файла из: " +
configFilePath, e);
        }
    }
}

```

Извлекает раздел batch конфигурации

Использует BatchConfig для построения BatchServiceConfiguration

Передаёт в строитель путь к файлу YAML (необработанные данные конфигурации)

При правильной структуре конфигурации, передаваемой в строитель облачного клиента, пакетному сервису не требуется выполнять специальную обработку. Если возникает ошибка, выдается исключение.

Проанализируем, как создание формата конфигурации YAML и встраивание в него структуры облачной клиентской библиотеки влияет на продукт. Первая проблема заключается в том, что мы создаем сильную связанность между сервисом и используемой облачной клиентской библиотекой, передавая путь к файлу конфигурации напрямую загрузчику облачной конфигурации. Тот ожидает, что в файле присутствует раздел **auth**. При отсутствии этого раздела выдается исключение. Если в будущем вы захотите перейти на другую облачную библиотеку, возникнут проблемы. Раздел **auth**, предоставляемый инструментом, становится контрактом (API). Если инструмент используется напрямую, клиенты программных систем должны предоставить файл YAML с конфигурацией аутентификации. Удалить или изменить раздел **auth** невозможно, если он стал не нужен или имеет другой формат.

Вторая проблема возникает, когда облачный клиент изменяется, устаревает или удаляет настройку конфигурации. Мы рассмотрим такие ситуации далее в этой главе.

Впрочем, у такого подхода есть свои преимущества. Если вы интегрируетесь с нижележащей системой, предоставляющей десятки и сотни настроек, прямая передача конфигурации может быть хорошим вариантом. Также важно, чтобы вызывающая сторона знала формат конфигурации следующей системы и применяла

его в своем коде. Наш сценарий использования удовлетворяет этому требованию, потому что облачный клиент разрешает вызывающей стороне передать настройки напрямую. Кроме того, вызывающая сторона в этом случае знает их структуру: файл YAML с разделом `auth`. Не нужно возиться с преобразованием настроек в правильную структуру, так что затраты на обслуживание отсутствуют.

В следующем разделе мы создадим стриминговый инструмент с другим подходом к конфигурации и UX используемого API. Настройки облачного клиента не будут раскрываться напрямую.

6.3. АБСТРАГИРОВАНИЕ НАСТРОЕК ЗАВИСИМОЙ БИБЛИОТЕКИ

Перейдем ко второму сервису, в котором используется иной подход к настройке зависимого облачного клиента. Стриминговый сервис, создаваемый в этом разделе, теперь предоставляет только принадлежащие ему настройки. Он использует их для конструирования облачного клиента, создание и конфигурация которого абстрагируются от пользователя. Конечные пользователи стримингового инструмента ничего не знают об облачном клиенте, используемом во внутренней реализации. В следующем листинге показана специфическая конфигурация стримингового сервиса, содержащая только одну настройку: `maxTimeMs`.

Листинг 6.14. Построение `StreamingServiceConfiguration`

```
public class StreamingServiceConfiguration {
    private final int maxTimeMs;

    public StreamingServiceConfiguration(int maxTimeMs) {
        this.maxTimeMs = maxTimeMs;
    }

    public int getMaxTimeMs() {
        return maxTimeMs;
    }
}
```

Значение `maxTimeMs` используется для хранения времени запроса в миллисекундах. Если время запроса превышает это значение, регистрируется предупреждение. В этом случае стриминговый сервис не объединяет запросы в пакеты, потому что низкая задержка обработки критична. Метод `loadData()` (как и предыдущий пакетный сервис) использует объект `Request`. Общее время обработки вычисляется вычитанием времени после загрузки облачного сервиса из начального времени. Рассмотрим логику из следующего листинга, которая проверяет, превышает ли общее время максимальное время из объекта конфигурации стримингового инструмента.

Листинг 6.15. Логика стримингового инструмента

```

public void loadData(Request request) {
    long start = System.currentTimeMillis();
    cloudServiceClient.loadData(request);
    long totalTime = System.currentTimeMillis() - start;
    if (totalTime > streamingServiceConfiguration.getMaxTimeMs()) {
        logger.warn(
            "Превышено время запроса! Время: {}, но оно должно быть
меньше чем: {}",
            totalTime,
            streamingServiceConfiguration.getMaxTimeMs());
    }
}

```

А теперь изучим механизм загрузки стриминговой конфигурации. Также будет показано, как абстрагируется конфигурация облачного сервиса.

6.3.1. Конфигурация стримингового сервиса

Стриминговый сервис также использует файл YAML для хранения конфигурации. Самое значительное отличие ее формата от конфигурации пакетной программы из раздела 6.2 состоит в том, что стриминговый сервис предоставляет все настройки в разделе **streaming**. Важно отметить, что стриминговый инструмент поддерживает только аутентификацию «имя пользователя/пароль». В следующем листинге приведен соответствующий раздел файла YAML.

Листинг 6.16. Конфигурация стримингового сервиса

```

streaming:
  username: u
  password: p
  maxTimeMs: 100
maxTimeMs: 100

```

Профильный раздел **streaming** определяет все настройки стримингового сервиса, которому принадлежит конфигурация. Соответственно, нашим клиентам ничего не известно об используемом клиенте облачного сервиса. Иначе говоря, конфигурация облачного сервиса абстрагируется от пользователя. Раздел **streaming** определяет четкий контракт, владельцем которого является стриминговый сервис. Такое решение выглядит проще с точки зрения UX, но требует обслуживания для отображения настроек из стримингового формата в формат облачного клиента.

В листинге 6.17 приведена логика создания стримингового сервиса. Все настройки, относящиеся к сервису, извлекаются из раздела **streaming**: инструмент читает значение **maxTimeMs** и конструирует **StreamingServiceConfiguration**. Самое важное — конструирование облачного клиента. Внутренняя облачная библиотека

и ее стратегия `UsernamePasswordAuthStrategy` абстрагируются от пользователя. Клиент `StreamingService` ничего не знает о механизме конфигурации. Кроме того, настройки имени пользователя и пароля применяются для конструирования `UsernamePasswordAuthStrategy`. Затем стратегия создает клиент облачного сервиса, используя его API программной конфигурации.

Листинг 6.17. Конструирование стримингового сервиса

```
public StreamingService create(Path configFilePath) {
    try {

        Map<String, Map<String, Object>> config =
            mapper.readValue(configFilePath.toFile(), yamlConfigType);
        Map<String, Object> streamingConfig =
            ➔ config.get("streaming");
        StreamingServiceConfiguration streamingServiceConfiguration =
            new StreamingServiceConfiguration((Integer)
                ➔ streamingConfig.get("maxTimeMs"));
        CloudServiceConfiguration cloudServiceConfiguration =
            new CloudServiceConfiguration(
                new UsernamePasswordAuthStrategy(
                    (String) streamingConfig.get("username"),
                    (String) streamingConfig.get("password")));
        return new StreamingService(
            streamingServiceConfiguration,
            new CloudServiceClientBuilder().create(cloudServiceConfiguration));
    } catch (IOException e) {
        throw new UncheckedIOException
            ➔ ("Ошибка при загрузке файла из: " + configFilePath, e);
    }
}
```

Этот раздел извлекается стриминговым сервисом и принадлежит ему

Использует `maxTimeMs` для создания конфигурации

Имя пользователя и пароль используются для конструирования конфигурации

Строитель с программным API, который создает облачный клиент

Важно отметить, что конструирование облачного клиента требует от нас дополнительной работы. Настройки, предоставленные стриминговым сервисом, нужно отобразить на облачную конфигурацию. После выпуска системы необходимо поддерживать это отображение, а это подразумевает дополнительные затраты на обслуживание. С другой стороны, UX конфигурации стримингового сервиса тоже улучшается, потому что вызывающим сторонам достаточно сосредоточиться на профильном разделе. Облачный клиент, используемый программой, скрывается за счет абстрагирования.

Предположим, вы интегрируетесь с нижележащей системой, предоставляющей десятки и сотни настроек. Важно, что мы выбираем вариант конфигурации, абстрагирующий эти настройки. Поэтому необходимо преобразовать каждую настройку нижележащей библиотеки в настройку сервиса. Для этого может понадобиться значительный объем кода, единственный смысл которого — перезапись настроек. Эффект на порядок усиливается, если у вас есть N сервисов или инструментов,

которые используют нижележащую систему, предоставляющую многие из этих настроек. В такой ситуации затраты на обслуживание весьма значительны.

В следующем разделе мы проанализируем обе конфигурации с точки зрения UX и затрат на обслуживание при добавлении в облачный клиент новой настройки. Вы увидите, что стоимость обслуживания приходится оплачивать заранее. К счастью, это дает определенные преимущества в долгосрочной перспективе. Проанализируем эти сценарии.

6.4. ДОБАВЛЕНИЕ НОВОЙ НАСТРОЙКИ ДЛЯ ОБЛАЧНОЙ КЛИЕНТСКОЙ БИБЛИОТЕКИ

Допустим, клиентский сервис изменяется и предоставляет новую настройку, отвечающую за тайм-аут. Она имеет профильный раздел `timeouts` в конфигурации YAML, как показано в следующем листинге.

Листинг 6.18. Добавление новой настройки тайм-аута

```
auth:
  strategy: username-password
  username: user
  password: pass

timeouts:
  connection: 1000
```

Новая настройка также добавляется в `CloudServiceConfiguration`. Реализация приведена в следующем листинге.

Листинг 6.19. Новая настройка тайм-аута для `CloudServiceConfiguration`

```
public class CloudServiceConfiguration {
    private final AuthStrategy authStrategy;
    private final Integer connectionTimeout;
    // Конструкторы, hashCode, равенство, методы чтения и записи не приводятся
}
```

Строитель облачного клиента извлекает раздел `timeouts` из конфигурации YAML и использует его для конструирования клиента. В следующем листинге приведена часть процесса добавления новой настройки для облачной клиентской библиотеки.

Листинг 6.20. Извлечение тайм-аута в `CloudServiceClientBuilder`

```
Map<String, Object> timeouts = config.get("timeouts");
// ...
return new DefaultCloudServiceClient(
    new CloudServiceConfiguration(authStrategy, (Integer)
timeouts.get("connection"))));
```

С точки зрения обеих программ (стриминговой и пакетной) это изменение важно, потому что оно не имеет обратной совместимости. И наоборот, если облачный клиент предоставляет значение по умолчанию, когда параметр *не* задан, изменение будет обладать обратной совместимостью. С другой стороны, если значение по умолчанию или передаваемое значение не задаются явно, новую версию облачного клиента сконструировать не удастся. Обе программы должны предоставить новое значение тайм-аута, чтобы иметь возможность сконструировать облачный клиент. Сначала проанализируем, как это изменение влияет на пакетный сервис, передающий настройки непосредственно строителю облачного клиента.

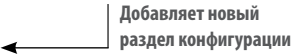
6.4.1. Добавление новой настройки в пакетный инструмент

Пакетный инструмент напрямую передает настройки от вызывающей стороны в строителя облачного клиента. Это означает, что клиент должен предоставить новый раздел `timeouts` для запуска пакетного инструмента. В следующем листинге показано, как должна выглядеть новая пакетная конфигурация YAML.

Листинг 6.21. Добавление нового раздела `timeouts`

```
auth:
  strategy: username-password
  username: u
  password: p

timeouts:
  connection: 1000
batch:
  size: 100
```



Добавляет новый
раздел конфигурации

Чтобы пакетный инструмент конструировал облачный клиент с новыми значениями конфигурации, все клиенты должны добавить этот раздел. Если они этого не сделают, то облачный клиент и конструирование пакетного сервиса, в котором он используется, завершится сбоем.

По поводу новой настройки необходимо сделать одно важное замечание. Как говорилось в разделе 6.2, `BatchServiceBuilder` передает файл YAML непосредственно в облачный клиент. Поэтому не требуется менять код для обработки нового параметра тайм-аута в пакетном инструменте. Необработанная конфигурация передается используемой облачной клиентской библиотеке, как показано на рис. 6.6.

Можно сделать вывод, что UX решения заметно не изменяется. Клиентам все равно придется конструировать облачный клиент и синхронизировать его с конфигурацией пакетного инструмента. Затраты на обслуживание инструмента близки к нулю, потому что для поддержки нового параметра изменений в коде не требуется. Передается необработанный файл, а `CloudServiceClientBuilder` извлекает из файла YAML разделы `auth` и `timeouts`.

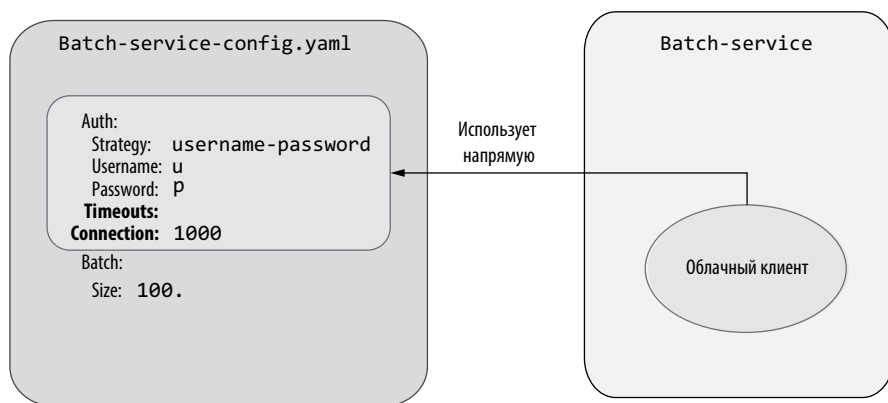


Рис. 6.6. Прямая передача настроек облачного клиента с новым разделом `timeouts`

Если предполагается, что изменения производятся довольно часто и накапливаются, то схема, представленная в этом разделе, эффективна. Кроме того, представьте, что несколько сервисов интегрируются с одним нижележащим облачным клиентом. Это означает, что при добавлении новой настройки ничего менять в этих сервисах не придется. Обязанность клиента — обеспечить обработку новых настроек. Бремя обслуживания распространяется на вызывающие стороны инструментов, поэтому можно считать, что в этом контексте UX будет не идеальным. Посмотрим, как добавить новые настройки облачного клиента в стриминговом сервисе.

6.4.2. Добавление новой настройки в стриминговый сервис

Стриминговый сервис применяет другой подход к UX и конфигурации используемой библиотеки: свои собственные настройки он предоставляет в специальном разделе `streaming`. Чтобы иметь возможность передать новую настройку тайм-аута соединения, мы должны добавить этот раздел в конфигурацию YAML стримингового сервиса, как показано в следующем листинге.

Листинг 6.22. Новая настройка тайм-аута для конфигурации стримингового сервиса

```
streaming:
  username: u
  password: p
  maxTimeMs: 100
connectionTimeout: 1000
```

Новая настройка предоставляется под именем `connectionTimeout`

Так как стриминговый инструмент конструирует облачный клиент на программном уровне, код, отвечающий за его создание, необходимо изменить. Для

этого значение `connectionTimeout` извлекается из файла YAML и передается `CloudServiceConfiguration`, как показано в следующем листинге.

Листинг 6.23. Новое значение тайм-аута в `StreamingServiceBuilder`

```
new CloudServiceConfiguration(  
    new UsernamePasswordAuthStrategy(  
        (String) streamingConfig.get("username"),  
        (String) streamingConfig.get("password"),  
        (Integer) streamingConfig.get("connectionTimeout"));
```

С каждой новой настройкой, вводимой в облачный клиент, связываются затраты на обслуживание. В реальных системах добавление конфигурации может требоваться чаще. Чем больше настроек появляется, тем выше связанные с ними затраты на обслуживание. Проанализируем этот механизм конфигурации в сценарии, при котором изменения происходят достаточно часто и накапливаются.

Каждая новая настройка облачного клиента должна отображаться на программном уровне. Если эта библиотека используется несколькими сервисами, придется менять код в каждом из них. Не забудьте, что каждое изменение кода влечет за собой дополнительные затраты на обслуживание. Кроме того, нужно обеспечить покрытие изменений сквозными тестами, а также провести высокоуровневую интеграцию. А когда качество изменений в обновленном коде становится достаточно высоким, измененное приложение необходимо развернуть в реальной среде.

И все это нужно повторить для каждого сервиса или программы, использующей клиентскую библиотеку! Чем больше сервисов, работающих с облачным клиентом, добавит новую настройку, тем больше вам придется сделать. Затраты на поддержку инкапсуляции будут достаточно высокими. Возможно, вы не увидите никакой пользы от этой дополнительной сложности.

В следующем разделе я приведу другой пример, который оправдывает стоимость обслуживания при этом подходе. Но для начала подведем итог тому, что вы узнали о UX и затратах на обслуживание обоих решений при добавлении настроек.

6.4.3. Сравнение UX-ориентированности и удобства обслуживания в двух решениях

Итак, вы увидели, что добавление новых настроек в используемую библиотеку привело к изменениям в механизмах конфигурации обоих инструментов и что:

- пакетный сервис распространяет изменения на сторону конечного пользователя;
- стриминговый сервис пытается абстрагировать факт использования облачного сервиса.

В пакетном сервисе клиент обязан предоставить новый профильный раздел с настройкой, которую ожидает получить облачный клиент. Важнейшее преимущество этого решения в том, что оно не предполагает затрат на обслуживание с нашей стороны. Изменять код не нужно, потому что файл конфигурации передается напрямую в строитель облачного клиента. Однако необходимо помнить, что затраты на обслуживание распространяются на конечного пользователя. Всем пользователям ваших сервисов и программ придется адаптировать свои конфигурации в соответствии с новыми настройками облачного клиента.

Так как стриминговый сервис абстрагирует свое использование облачного сервиса, он должен отобразить настройки, предоставляемые конечными пользователями, на конфигурацию облачного клиента. Добавление новой настройки в облачный клиент также требует добавления этой настройки в стриминговый сервис, который является ее владельцем. Конечный пользователь ничего не знает о скрытой от него конструкции облачного сервиса. И это имеет свою стоимость. С другой стороны, каждую новую настройку используемой системы необходимо сделать доступной и сопоставить с ожидаемым форматом стримингового сервиса. Это требует затрат на обслуживание: приходится изменять код стримингового сервиса.

В табл. 6.1 приведена краткая сводка двух сценариев. Важно отметить, что UX и затраты на обслуживание представлены в контексте одного сервиса.

Таблица 6.1. Добавление новой настройки в клиент и ее влияние на сервисы

Название	Затраты на обслуживание	UX
Пакетный сервис	Затраты отсутствуют	Пользователь должен добавить новую настройку
Стриминговый сервис	Растущие затраты	Пользователь должен добавить новую настройку

Если вы используете облачный клиент из N сервисов, эти затраты умножаются на N . Затраты на обслуживание растут для каждой программы, которая использует облачный клиент и инкапсулирует настройки.

В следующем разделе рассмотрена ситуация, в которой абстракция стриминговой программы стоит дополнительных затрат. Облачный клиент устареет, и его настройки потребуются удалить.

6.5. УДАЛЕНИЕ НАСТРОЕК В ОБЛАЧНОЙ КЛИЕНТСКОЙ БИБЛИОТЕКЕ

В этом разделе анализируется сценарий, в котором может возникнуть необходимость в удалении настроек облачного клиента. Как вы, должно быть, помните,

облачный клиент использует определенную стратегию аутентификации при подключении к облачному сервису. Допустим, через некоторое время вы видите, что текущая стратегия `UsernamePasswordAuthStrategy` небезопасна, потому что пароль в виде простого текста хранится в конфигурации YAML и в памяти, что довольно рискованно. Злоумышленник может похитить пароли, используемые в коде.

Стоит разработать новую стратегию, `UsernamePasswordHashedAuthStrategy`, где применяется хешированная версия пароля при проведении аутентификации. Для этого используется алгоритм SHA-256 из класса, реализующего хеширование (<http://mng.bz/q2aA>). При аутентификации запроса сравниваются хешированные версии паролей. В следующем листинге приведен код новой стратегии аутентификации с хешированием.

Листинг 6.24. Новая стратегия `UsernamePasswordHashedAuthStrategy`

```
public class UsernamePasswordHashedAuthStrategy implements AuthStrategy {
    private final String username;
    private final String passwordHash;
    public UsernamePasswordHashedAuthStrategy(String username, String
passwordHash) {
        this.username = username;
        this.passwordHash = passwordHash;
    }
    @Override
    public boolean authenticate(Request request) {
        if (request.getUsername() == null || request.getPassword() == null) {
            return false;
        }
        return request.getUsername().equals(username)
            && toHash(request.getPassword()).equals(passwordHash);
    }
    public static String toHash(String password) {
        return Hashing.sha256().hashString(password,
            StandardCharsets.UTF_8).toString();
    }
}
```

Пароль сохраняется в хешированной форме

Выполняет аутентификацию с хешированной версией пароля

Использует алгоритм SHA-256 для хеширования пароля

Как видно из следующего листинга, новой стратегии аутентификации присваивается идентификатор `username-password-hashed`. Клиенты облачной конфигурации должны использовать его вместо прежнего значения `username-password`, которое конструирует версию с простым текстовым паролем.

Листинг 6.25. Запрет прежней стратегии аутентификации

```
public class CloudServiceClientBuilder {
    private static final String USERNAME_PASSWORD_STRATEGY = "username-password";
    private static final String TOKEN_STRATEGY = "token";
    private static final String
    USERNAME_PASSWORD_HASHED_STRATEGY = "username-password-hashed";
}
```

Реализует новую стратегию `username-password-hashed`

```
// ...
public DefaultCloudServiceClient create(Path configFilePath) {
    // ...
    if (authConfig.get("strategy").equals(USERNAME_PASSWORD_HASHED_STRATEGY))
    {
        authStrategy =
            new UsernamePasswordHashedAuthStrategy(
                (String) authConfig.get("username"), (String)
                authConfig.get("password"));
        // Конструирует UsernamePasswordHashedAuthStrategy

    } else if (authConfig.get("strategy").equals(TOKEN_STRATEGY)) {
        authStrategy = new TokenAuthStrategy((String) authConfig.get("token"));
    } else if (authConfig.get("strategy").equals(USERNAME_PASSWORD_STRATEGY))
    {
        throw new UnsupportedOperationException(
            "The " + USERNAME_PASSWORD_STRATEGY + " strategy is no longer
            supported.");
        // Если задана стратегия username-password,
        // выдается исключение
    }
    return new DefaultCloudServiceClient(
        new CloudServiceConfiguration(authStrategy, (Integer)
        timeouts.get("connection")));
}
}
```

Когда облачный сервис конструирует стратегию аутентификации, для прежней стратегии `username-password` выдается исключение, которое сообщает, что стратегия больше не поддерживается. Это означает, что все вызывающие стороны должны перейти на новую стратегию, если хотят пользоваться облачным клиентом. Посмотрим, как такое изменение поведения влияет на пакетный инструмент.

6.5.1. Удаление настройки из пакетного инструмента

Как вы уже знаете, пакетный сервис передает файл YAML, полученный от клиента, непосредственно в облачный клиент. До сих пор все клиенты применяли стратегию с именем пользователя/паролем или маркером безопасности. Если клиент задает устаревшую стратегию `username-password`, конфигурация пакетного сервиса выдает исключение. Теперь все клиенты должны переключиться на новый тип, если они хотят использовать пакетный сервис. Это значительно влияет на UX решения.

Все клиенты, настраивающие пакетное решение, сталкиваются с проблемой аутентификации облачного клиента. Это поведение можно наблюдать в модульном тесте, который использует файл `batch-service-config-timeout.yaml` со стратегией `username-password`. Код теста приведен в следующем листинге.

Листинг 6.26. Выдача исключения для неподдерживаемой стратегии аутентификации

```

@Test
public void shouldThrowIfUsingNotSupportedAuthStrategy() {
    // Дано
    Path path =
        Paths.get(
            Objects.requireNonNull(
                getClass().getClassLoader().getResource("batch-service-
config-timeout.yaml"))
                .getPath());

    // Когда
    assertThatThrownBy(() -> new BatchServiceBuilder().create(path))
        .isInstanceOf(UnsupportedOperationException.class)
        .hasMessageContaining("Стратегия username-password strategy больше
не поддерживается.");
}

```

Теперь все клиенты видят исключение `UnsupportedOperationException`. Это означает, что все клиенты пакетного инструмента должны будут преобразовать свою конфигурацию YAML на новый режим `user-name-password-hashed`! UX такого решения оставляет желать лучшего. Мы раскрываем внутреннее устройство сторонней библиотеки, так что каждое изменение этой конфигурации потребует адаптации клиентского кода.

Представьте сценарий, в котором пакетный сервис используется несколькими клиентскими системами. Мы публикуем новый пакетный сервис, который не позволяет конечным пользователям применять стратегию с именем пользователя и паролем. Когда конечные пользователи изменят свое ПО для нового пакетного сервиса, они не смогут развернуть его без изменений в конфигурации YAML. Каждый клиент, оставивший режим аутентификации без изменений, получит исключение во время выполнения при использовании новой версии пакетного сервиса. Для обеспечения обратной совместимости и уменьшения количества проблем с UX приходится создавать *неуклюжее* обходное решение.

Сначала необходимо загрузить файл конфигурации из `configFilePath`. Мы просматриваем содержимое файла и ищем запись для `auth.strategy`. Найдя ее, мы меняем стратегию конфигурации, поставив `username-password-hashed` вместо `username-password`. Затем необходимо извлечь простой текстовый пароль и вручную хешировать его, заменяя запись в карте. Обходное решение представлено в следующем листинге.

Листинг 6.27. Обходное решение `BatchServiceBuilder`

```

// НЕ ДЕЛАЙТЕ ТАК
public BatchService create(Path configFilePath) {

```



```

try {
    Map<String, Map<String, Object>> config =
        mapper.readValue(configFilePath.toFile(), yamlConfigType);
    Map<String, Object> batchConfig = config.get("batch");
    BatchServiceConfiguration batchServiceConfiguration =
        new BatchServiceConfiguration((Integer) batchConfig.get("size"));

    Map<String, Object> authConfig = config.get("auth");
    if (authConfig.get("strategy").equals(USERNAME_PASSWORD_STRATEGY)) {
        authConfig.put("strategy",
            USERNAME_PASSWORD_HASHED_STRATEGY);
    }
    String password = (String) authConfig.get("password");
    String hashedPassword = toHash(password);
    authConfig.put("password", hashedPassword);
    Path tempFile = Files.createTempFile(null, null);
    Files.write(tempFile, mapper.writeValueAsBytes(config));

    CloudServiceClient cloudServiceClient = new
        CloudServiceClientBuilder().create(tempFile);
    return new BatchService(batchServiceConfiguration, cloudServiceClient);
} catch (IOException e) {
    throw new UncheckedIOException
        ("Проблема при загрузке файла из: " + configFilePath, e);
}

```

Первая утечка абстракции конфигурации облачного сервиса

Переопределение настроек может привести к трудноустранимым ошибкам!

Еще одна утечка конфигурации

Переопределяется другая настройка

Создает временный файл

Сохраняет модифицированную конфигурацию

Передает измененный файл конфигурации (не тот, который передается вызывающей стороной)

Наконец, нужно сохранить измененную конфигурацию в новом временном файле и передать путь к этому файлу `CloudServiceClientBuilder`. Такое решение просто ужасно: оно изменяет исходный файл и параметр конфигурации (без ведома пользователя) и может ввести баги, которые очень трудно отладить. Кроме того, при каждом построении клиента приходится создавать временный файл.

Важно заметить, что реальные имена параметров конфигурации утекают из `CloudServiceClientBuilder` в новое неуклюжее обходное решение `BatchServiceBuilder`, что приводит к появлению сильной связанности между компонентами. Вдруг оказывается, что строитель службы, который обычно отвечает только за загрузку разделов конфигурации из файла YAML, должен знать точную структуру конфигурации облачного клиента и изменять ее.

Стриминговый сервис использует другой подход к конфигурации. В следующем разделе показано, как в таком сервисе решается проблема удаления настроек.

6.5.2. Удаление настроек из стримингового сервиса

В стриминговом инструменте внутренняя стратегия аутентификации, используемая облачной библиотекой, абстрагируется от пользователя. Клиентам

сервиса ничего не известно о его механизме конфигурации. Можно прозрачно изменить стратегию аутентификации, при этом пользователь об этом не узнает, а совместимость не будет нарушена. На новые стратегии можно переходить, не создавая проблем для пользователей, а конфигурация YAML стримингового сервиса никак не изменится.

В листинге 6.28 используется то же имя пользователя и пароль, что и ранее, при этом пароль передается в виде простого текста. `StreamingServiceBuilder` конструирует объект `UsernamePasswordHashedAuthStrategy`, а затем передает ему хешированную версию пароля.

Листинг 6.28. Абстрагирование конструирования стратегии с хешированием

```
CloudServiceConfiguration cloudServiceConfiguration =
    new CloudServiceConfiguration(
        new UsernamePasswordHashedAuthStrategy(
            (String) streamingConfig.get("username"),
            toHash((String) streamingConfig.get("password")),
            (Integer) streamingConfig.get("connectionTimeout"));
```

Конструирует стратегию с хешированием (вместо стратегии с простым текстом)

Пароль хешируется до передачи в стратегию с хешированием

Изменение поведения скрыто от пользователей стримингового инструмента. Такое решение обеспечивает более высокий уровень UX, потому что оно не требует смены конфигурации стримингового сервиса. Конечные пользователи сервиса смогут легко использовать новую версию, ничего не меняя на своей стороне.

Стриминговый сервис может сменить облачную клиентскую библиотеку незаметно для конечных пользователей. Команда разработчиков сервиса легко переведет его на другую библиотеку при необходимости. Отображение настроек уже реализовано, так что в такой ситуации к новому формату конфигурации нужно будет адаптировать только слой отображения.

Незаметное для конечного пользователя отображение прежней стратегии аутентификации на новую стратегию `UsernamePasswordHashedAuthStrategy` может быть хорошим временным решением. Тем не менее в долгосрочной перспективе следует мигрировать на `UsernamePasswordHashedAuthStrategy`, потому что она обеспечивает более высокую безопасность для пользователей.

В какой-то момент придется заняться реализацией миграции, но поскольку стриминговый инструмент инкапсулирует нижележащую облачную конфигурацию, процесс перехода упрощается. Например, можно добавить в конфигурацию новое свойство с хешированным паролем. Если конечный пользователь предоставляет такой пароль, уже не требуется вручную преобразовывать простой текстовый пароль в хешированный. Вместо этого можно использовать новую стратегию `UsernamePasswordHashedAuthStrategy`.

В процессе миграции можно поддерживать оба способа предоставления паролей. Это позволит клиентам стримингового сервиса мигрировать, не беспокоясь о нарушении изменений используемого облачного клиента.

В следующем разделе сравниваются UX-ориентированность и затраты на обслуживание для двух решений.

6.5.3. Сравнение UX-ориентированности и затрат на обслуживание для двух решений

Из всего сказанного можно сделать вывод, что UX-ориентированность и затраты на обслуживание для двух решений будут разными в случае удаления настроек в нижележащей системе. Рассмотрим эти различия.

Во-первых, стриминговый инструмент является владельцем всей конфигурации, и миграция всех нижележащих компонентов в нем создает меньше проблем. Если вы захотите убрать облачный клиент и заменить его другим, в стриминговом сервисе это сделать проще.

Для этого сценария прямая передача пакетным сервисом настроек от клиентов к облачному клиенту совсем не подходит. Удаление зависимых настроек означает, что всем клиентам придется одновременно переходить на новое значение. Скрыть это обстоятельство от них не удастся. Кроме того, UX системы получается слишком хрупким. Чтобы справиться с проблемой, придется сконструировать неуклюжее обходное решение с высоким риском ошибок.

Дополнительная абстракция конфигурации, введенная в стриминговом сервисе, дает возможность совершенствовать инструменты так, чтобы это было удобно для пользователя. Завершим обсуждение табл. 6.2, в которой сравниваются два подхода.

Таблица 6.2. Удаление настроек из клиента и его влияние на два инструмента

Название	UX	Затраты на обслуживание
Пакетный сервис	Низкий уровень; значительное влияние на пользователей	Высокие/неприемлемые
Стриминговый сервис	Высокий уровень; не влияет на пользователей	Очень низкие

Решение, касающееся затрат на обслуживание, связывается с риском критических, несовместимых изменений нижележащего компонента. Если облачная клиентская библиотека развивается с нарушением обратной совместимости, сервисы, использующие клиент, должны абстрагировать свой механизм конфигурации. Допустим, вы разрабатываете сервис и можете повлиять на жизненный цикл библиотеки. Возможно, при этом вам удастся сократить количество

изменений, нарушающих обратную совместимость. Вы даже можете запретить их и развивать библиотеку без критических изменений. Тогда вам не придется нести дополнительные затраты на обслуживание, связанные с абстрагированием его механизма конфигурации.

С другой стороны, предположим, что вы используете библиотеку, которая развивается непредсказуемым образом, и не влияете на ее жизненный цикл. Из-за этого могут возникнуть изменения, нарушающие обратную совместимость, и от них следует защититься. В таком сценарии дополнительные затраты на обслуживание оправданны. Это существенно улучшит UX программы, которая будет удобной для клиентов.

В этой главе были представлены разные способы проектирования программных инструментов. Мы начали с создания облачной клиентской библиотеки, которую затем использовали в двух инструментах: стриминговом и пакетном. Для первого из них мы выбрали непрямой подход к управлению конфигурацией. Она абстрагируется от нижележащей библиотеки, а добавление новых настроек требует затрат на обслуживание. Во втором варианте используется API конфигурации облачного клиента без прослойки абстракции. Он позволяет развивать пакетный сервис без затрат на обслуживание при добавлении настроек в используемую библиотеку.

Ситуация кардинально меняется при удалении настроек нижележащих компонентов. Абстракция, введенная в стриминговом сервисе, позволила обеспечить превосходный уровень UX и низкие затраты на обслуживание. С другой стороны, пакетный сервис не мог обработать это изменение при сохранении хорошего уровня UX. Небольшие затраты на обслуживание улучшили UX в этой ситуации. В следующей главе рассматриваются основные компромиссы и ошибки, встречающиеся при работе с API даты и времени.

ИТОГИ

- Технические решения могут влиять на UX.
- Добавление новых настроек в используемые библиотеки может происходить с нулевыми затратами на обслуживание.
- Дополнительная абстракция позволяет развивать программные инструменты без нарушения совместимости. С другой стороны, она увеличивает затраты на обслуживание.
- С дополнительной абстракцией каждое изменение в используемом компоненте требует большей работы кода.
- Если продукт ориентирован на потребителя и уровень UX для вас важен, лучше не раскрывать внутренние подробности библиотек, используемых в коде.



Эффективная работа с датой и временем

https://t.me/it_boooks

В этой главе:

- ✓ Данные даты и времени в определенных концепциях.
- ✓ Ограничение области действия и документирование точных требований к продукту.
- ✓ Выбор лучших библиотек для использования в коде с датой и временем.
- ✓ Последовательное применение концепций даты и времени в коде и обеспечение тестируемости кода времени и даты.
- ✓ Выбор текстовых форматов для даты и времени.
- ✓ Граничные случаи, связанные с календарными вычислениями и часовыми поясами.

Дата и время совершенно естественно используются почти во всех приложениях, даже если это просто метка времени в сообщениях журнала. К сожалению, часто функции даты и времени создают большие проблемы — либо за счет чрезмерного усложнения кода, либо из-за багов, которые проявляются всего на два часа в год или только у пользователей какого-нибудь удаленного уголка планеты. Игнорировать такие баги проще простого, но с правильными инструментами их можно избежать.

Такие инструменты делятся на две категории:

- *Концепции* помогают обдумать и четко описать информацию, с которой вы работаете.
- *Библиотеки* помогают преобразовать концепции в код.

Иногда используемые библиотеки являются частью платформы (например, `java.time`, появившаяся в Java 8), иногда это сторонние библиотеки, которые должны устанавливаться явно, как Noda Time для .NET (*абсолютно* случайный выбранный пример, а может и не совсем случайно. Джон является основным автором Noda Time).

В зависимости от того, с какой платформой и библиотеками вы работаете, может оказаться, что между концепциями, описанными в этой главе, и типами, которые вы используете для их представления, нет однозначного соответствия. Это нормально. Такое несоответствие *немного* усложняет жизнь, но концепции все равно применимы к проекту; просто нужно тщательнее прописывать свои намерения в комментариях при выборе имен или написании документации (или везде и сразу).

Помимо описания концепций и возможностей их практического применения в коде, эта глава содержит рекомендации о том, как организовать эффективное тестирование кода, связанного с датой или временем. Прочитав эту главу, вы сможете грамотно и уверенно проектировать и реализовывать логику даты и времени.

Для максимальной наглядности возьмем пример интернет-магазина. На рис. 7.1 представлены требования к продукту в том виде, в котором их можно передать команде разработчиков.

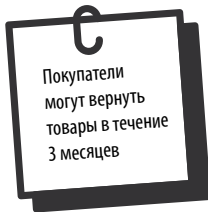


Рис. 7.1. Высокоуровневые требования для сайта интернет-магазина

Читая эту главу, вы увидите, как данное требование преобразуется в другое, намного более детализированное и имеющее четко тестируемые критерии. Затем мы реализуем требование и напишем соответствующие тесты. Но начать стоит с рассмотрения концепций, почти без упоминания кода.

7.1. КОНЦЕПЦИИ ПРЕДСТАВЛЕНИЯ ДАТЫ И ВРЕМЕНИ

Как это часто бывает, в тему даты и времени можно погружаться бесконечно, находя все более интересные примеры нетипичного поведения. Но так можно никогда не вернуться на поверхность. Некоторые платформы и библиотеки выбирают иной (и тоже неверный) курс — пытаются сделать вид, что все очень просто, и упускают действительно важные случаи. Мы адаптировали концепции, представленные в этом разделе, для достижения «золотой середины»: они достаточно подробны для большинства коммерческих приложений, но не настолько глубоки, чтобы эта глава заняла целую книгу.

ПРИМЕЧАНИЕ

Если вы работаете в узкоспециализированных областях, вам придется искать информацию где-то еще, но даже в этом случае рассмотренных концепций может быть достаточно для *большинства* ваших приложений. Если вы проектируете устройства GPS, занимаетесь представлением данных античной истории или пишете клиент NTP (Network Time Protocol), изложенных концепций также будет достаточно в *большинстве* случаев. Старайтесь по возможности ограничить и локализовать наиболее сложные аспекты, требующие усилий.

Кроме того, если вы отлично разбираетесь в принципах работы корректировочных секунд, некоторые натяжки в рассуждениях вам могут не понравиться. Я вас понимаю, но это одна из тех областей, в которых *абсолютная* точность только мешает ясности изложения.

Там, где концепции имеют прямое представление в библиотеках `java.time` и `Noda Time`, будут приводиться соответствующие типы, чтобы при желании вы могли продолжить экспериментировать с ними. Начнем с некоторых базовых концепций: момента времени, эпохи и интервалов.

7.1.1. Машинное время: моменты времени, эпохи и интервалы

Приемы работы с информацией даты и времени очень сильно зависят от конкретной культуры. К примеру, религия, вероятно, влияет на эту область программирования сильнее всего. Хотя понимать этот культурный аспект очень важно, также полезно попытаться по возможности исключить его из вычислений. Вот почему мы начнем с рассмотрения более *чистых* и понятных концепций, где нет никакой путаницы, которую люди пытаются привнести в программирование.

Момент времени

Тип в `java.time`: `java.time.Instant`. Тип в `Noda Time`: `NodaTime.Instant`.

Момент времени — универсальная метка времени. Два человека, находящихся в любых точках мира (или за его пределами!), могут договориться о том, что означает *сейчас* как момент времени. Они могут посмотреть на часы и увидеть разное местное время из-за разных часовых поясов или не согласиться с названием месяца из-за культурных особенностей, но момент времени они все равно поймут одинаково. Момент можно рассматривать как своего рода *машинное время*, которое не зависит от суетных человеческих концепций: *дней, лет* и т. д.

Моменты можно рассматривать как точки временной оси, неделимые на меньшие части. Пример этой концепции показан на рис. 7.2.

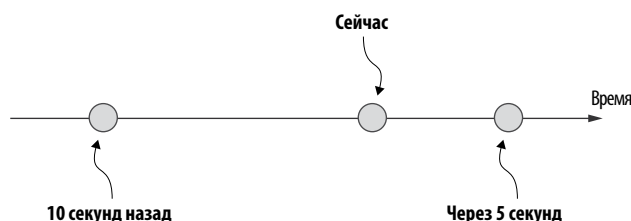


Рис. 7.2. Ось неделимых моментов времени

Представление момента времени может стать невероятно сложным (и даже невозможным), если учитывать теорию относительности и другие заковыристые понятия физики. Это один из тех случаев, в которых стремление добиться *абсолютного соответствия законам физической Вселенной* ошибочно почти всегда.

Моменты времени естественным образом подходят для описания ситуации, *когда что-то произошло* — например, была закреплена транзакция базы данных или создана запись в журнале. Возможно, вам интересно внутреннее представление момента времени. Хотя оно является деталью реализации, его все равно полезно рассмотреть, но для этого потребуется новая концепция.

Эпоха

Ось времени на рис. 7.2 не имеет числовых меток; точки размещаются относительно друг друга. Стандартное решение — договориться об искусственной *нулевой точке*, называемой *началом эпохи*, и отсчитывать все значения от нее. Добавим начало эпохи в существующий пример, в котором эпоха начинается за 15 с до *текущего момента*. На этой стадии каждый момент времени можно выразить количеством секунд от начала эпохи. На рис. 7.3 приведено графическое представление этой концепции: к рис. 7.2 добавляется начало эпохи и относительные промежутки времени от начала эпохи.

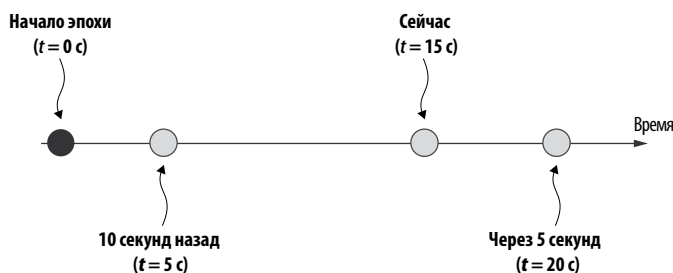


Рис. 7.3. Ось времени с началом эпохи

Здесь важно то, что для всех, кто использует одно представление, начало эпохи одинаково, так что в некоторых отношениях мы просто немного сместили проблему. Все равно необходимо договориться об одном моменте времени, но это позволит представить любой момент на оси.

В большинстве систем используется *начало эпохи Unix* — момент времени, приходящийся на полночь 1 января 1970 года по времени UTC. UTC, месяцы или годы мы еще не обсуждали; у всех описаний даты и времени есть одна проблема — концепции часто кажутся циклическими.

Впрочем, это не единственное начало эпохи, часто встречающееся на практике. Начало эпохи .NET приходится на полночь 1 января 1 года нашей эры (AD 1), хотя AD 1 — пролептический григорианский календарь, добавляющий *еще больше* сложности, о которой мы еще не говорили.

В Excel и представлении Microsoft COM начало эпохи приходится на начало 1900 года, хотя такие эпохи сложнее обсуждать из-за ошибок в программных продуктах, которые считают 1900 год високосным.

В хорошо инкапсулированных библиотеках даты и времени не нужно знать, какая эпоха используется во внутреннем представлении, хотя многие из них содержат функции для преобразования между представлением библиотеки и, например, количеством секунд от начала эпохи Unix. По этой причине в библиотеках даты и времени обычно даже не виден тип, инкапсулирующий концепцию эпохи.

В предыдущих примерах рассматривалось время от начала эпохи в секундах, но, конечно, в реальной жизни часто требуются более точные измерения времени. Вместо того чтобы всегда задавать конкретную единицу времени, полезно инкапсулировать концепцию *продолжительности прошедшего времени* в виде *промежутка*.

Промежуток времени

Тип в java.time: `java.time.Duration`. Тип в Noda Time: `NodaTime.Duration`.

Промежуток определяет *прошедшее* время, а не *точку* во времени. Он рассчитывается как разность между двумя точками на оси времени. Если запустить секундомер, то на нем будет отображаться промежуток. Он может быть как положительным, так и отрицательным (например, момент, наступивший до начала эпохи, во внутренней реализации может представляться отрицательным промежутком). На логическом уровне поддерживаются следующие операции с моментами и промежутками:

- Момент - Момент => Промежуток
- Промежуток + Промежуток => Промежуток
- Момент + Промежуток => Момент
- Момент - Промежуток => Момент

На рис. 7.4 эти операции представлены графически. В частности:

- Результатом *сейчас* - *x* является промежуток продолжительностью 10 секунд.
- Результатом 10 секунд + 5 секунд является промежуток продолжительностью 15 секунд.
- Промежутки можно складывать и вычитать, чтобы получить 10 секунд до текущего момента или 5 секунд после текущего момента.

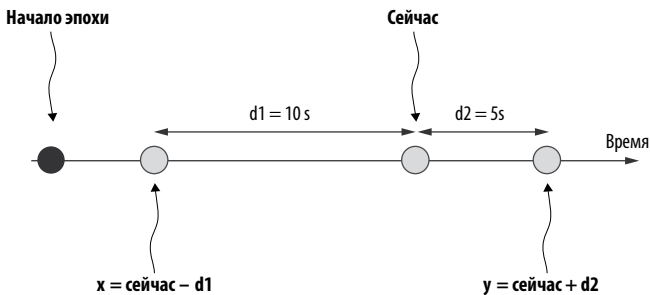


Рис. 7.4. Вычисления с моментами и промежутками на временной оси

Точность внутреннего представления промежутка обычно ограничена. Как правило, на практике задается точность до миллисекунд, микросекунд или наносекунд, а также тактов; такт используется в Windows и .NET, один такт равен 100 нс.

Важно то, что промежуток всегда жестко фиксирован продолжительностью измеряемого времени. Таким образом, 1 секунда, 5 микросекунд и 3 часа являются допустимыми промежутками, а 2 месяца — нет, потому что продолжительность месяца меняется. Является ли 1 день допустимым промежутком? Это зависит

от того, как именно понимать «1 день». Если рассматривать его как *время, прошедшее от полуночи одного дня до полуночи следующего* — нет, поскольку здесь мы заходим на территорию часовых поясов, где продолжительность дня может составлять 23 часа или 25 часов. Но если рассматривать 1 день как синоним для 24 часов, тогда он *является* допустимым промежутком.

ПРИМЕЧАНИЕ

Если так будет проще, можно рассматривать момент времени как аналог точки в геометрии, а промежуток — как аналог вектора. Все операции, поддерживаемые для моментов времени и промежутков, соответствуют операциям с точками и векторами.

В прошлом некоторые библиотеки избегали инкапсуляции концепции промежутков; вместо этого числа и единицы хранились раздельно. Это приводило к появлению сигнатур функций следующего вида (из `java.util.concurrent.locks.Lock`):

```
boolean tryLock(long time, TimeUnit unit)
```

Хотя в некоторых случаях такое разделение полезно, обычно оно получается слишком громоздким по сравнению с типом `Duration`, который может использоваться везде, где актуальна концепция прошедшего времени.

Несмотря на то что моменты времени и промежутки являются важнейшими концепциями машинного времени (а начало эпохи — своего рода вспомогательной концепцией), библиотеки часто предоставляют дополнительные типы для большего удобства. Самый популярный из них — *интервал*, инкапсулирующий два момента времени: начало и конец. Библиотеки по-разному определяют, могут ли интервалы быть открытыми (без начального или конечного момента) и может ли начало приходиться на более позднее время, чем конец (своего рода *отрицательный интервал*). Желательно знать, какие возможности поддерживает используемая библиотека, но мы не будем углубляться в подробности, потому что интервал не настолько фундаментальная концепция, как момент времени или промежуток.

Машинное время часто бывает полезным, но его формат очень неудобен для конечных пользователей и даже разработчиков. Представьте, что вы читаете файл журнала; что бы вы предпочли увидеть: `1605255526` или `2020-11-13T08:19:46Z`? Вычисления с участием человека часто заметно усложняются, и это особенно справедливо для даты и времени. Посмотрим, как люди привыкли делить время.

7.1.2. Календарные системы, даты, время и периоды

Если вы когда-нибудь сталкивались с багами часовых поясов, вас может удивить отсутствие часовых поясов в списке понятий, описываемых в этом разделе. Не беспокойтесь, дойдет дело и до них — просто еще не время. Сначала представим

мир без часовых поясов и разберемся, почему он не так прост, как можно ожидать. Начнем с вопроса, ответ на который кажется тривиальным: какой сегодня день?

Календарные системы: деление времени на дни, месяцы и годы

Типы в `java.time`: `java.time.chrono.Chronology`, `java.time.LocalDate` и `java.time.chrono.ChronoLocalDate`. Типы в `Noda Time`: `NodaTime.CalendarSystem` и `NodaTime.LocalDate`.

Один из универсальных аспектов человеческой жизни заключается в том, что она делится на дни. В каждой цивилизации существует понятие дня и ночи, обычно мы работаем днем и спим ночью. Таким образом, деление оси времени на дни выглядит абсолютно естественно.

Времена года исторически важны для большей части человечества, и хотя в наши дни земледельцев стало меньше, годичный цикл все еще влияет на нашу жизнь, поэтому деление оси времени на годы тоже разумно.

Месяцы — скорее удобство, чем полезный уровень детализации. Цикл фаз Луны продолжительностью 29,5 дня, вероятно, когда-то много значил для цивилизаций и разработки календарных систем, но сейчас его влияние на жизнь человека гораздо меньше.

Таким образом, *календарная система* представляет собой способ обозначения конкретного дня в контексте года, месяца этого года и дня этого месяца. Если бы существовала только одна календарная система, то задача решалась бы относительно просто, но на практике это не так.

ПРИМЕЧАНИЕ

Дата и время, рассматриваемые с бытовой точки зрения — с днями, месяцами и годами, — называются *гражданским временем* (civil time). Гражданское время сильно зависит от культуры в отличие от машинного времени, которое рассматривалось в предыдущем разделе.

Вернемся к вопросу, который привел нас к этому разделу: я пишу эти строки 20 ноября 2020 года (по крайней мере, в Великобритании). На первый взгляд это вполне однозначное утверждение, но даже в нем содержится неявное утверждение, ведь я также могу достаточно точно утверждать, что сегодня 7 ноября 2020 года. Как одному дню могут соответствовать две даты одновременно? Дело в том, что сегодня 20 ноября 2020 года по григорианскому календарю и 7 ноября — по юлианскому. Также сегодня 4-й день месяца кислев 5781 года по древнееврейскому календарю и 4-й день месяца раби ас-сани 1442 года по исламскому календарю хиджры. И это лишь некоторые из календарных систем, используемых в мире. В табл. 7.1 приведены некоторые дни, близкие к сегодняшней дате в этих системах.

Таблица 7.1. Даты в четырех календарных системах

Григорианский календарь	Юлианский календарь	Древнееврейский календарь	Календарь хиджры
16 ноября 2020 года	3 ноября 2020 года	29 хешван 5781 года	30 раби аль-авваль 1442 года
17 ноября 2020 года	4 ноября 2020 года	1 кислев 5781 года	1 раби ас-сани 1442 года
18 ноября 2020 года	5 ноября 2020 года	2 кислев 5781 года	2 раби ас-сани 1442 года
19 ноября 2020 года	6 ноября 2020 года	3 кислев 5781 года	3 раби ас-сани 1442 года
20 ноября 2020 года	7 ноября 2020 года	4 кислев 5781 года	4 раби ас-сани 1442 года
21 ноября 2020 года	8 ноября 2020 года	5 кислев 5781 года	5 раби ас-сани 1442 года
22 ноября 2020 года	9 ноября 2020 года	6 кислев 5781 года	6 раби ас-сани 1442 года

Календарные системы могут очень сильно различаться. Григорианский календарь почти идентичен юлианскому; отличие лишь в том, какие годы считаются високосными. Сравните с древнееврейской календарной системой, в которой длина месяцев хешван и кислев изменяется от года к году, а *високосным* считается год, содержащий не лишний день, а лишний месяц (месяц адар разбивается на адар I и адар II). С исламом связано много календарных систем, из-за чего довольно трудно определить, о какой из них идет речь, если кто-то говорит просто об *исламском календаре*.

Но больше всего с точки зрения кода меня удивил календарь Бади, используемый у бахаистов. В нем каждый год содержит 19 месяцев, которые состоят из 19 дней, и 4 или 5 дней, приходящихся между 18-м и 19-м месяцем. Эти дни вообще не принадлежат никакому месяцу.

Наконец, все вышесказанное предполагает, что все согласны с тем, когда заканчивается один день и начинается другой — в полночь, не так ли? Но это справедливо не для всех календарных систем. В древнееврейских и исламских календарях (среди прочего) границей между днями считается закат, а не полночь.

Я понимаю, что все это звучит ужасно, но, как было показано в разделе 7.2.1, в большинстве случаев вам почти не придется беспокоиться об этих различиях. Это была хорошая новость; а плохая в том, что, даже придерживаясь григорианского календаря, нужно соблюдать осторожность. Но если ось времени (еще раз: без часовых поясов) разделена на годы, месяцы и дни, сослаться на время суток будет относительно несложно.

Время суток

Тип в `java.time`: `java.time.LocalDateTime`. Тип в `Noda Time`: `NodaTime.LocalDateTime`.

И хотя *существуют* системы, в которых используются разные единицы времени, скорее всего, в большинстве случаев их можно игнорировать. Если вы захотите узнать больше, начать можно со статьи «Internet Time» на сайте Swatch (<https://www.swatch.com/en-us/internet-time.html>).

Если вынести все это за скобки и отложить в сторону часовые пояса и корректировочные секунды, можно принять, что день состоит из 24 часов, каждый час состоит из 60 минут, а каждая минута — из 60 секунд. Секунды можно и дальше делить на нужные единицы: миллисекунды, микросекунды или наносекунды.

Да, все *почти* так просто, поэтому этот подраздел будет самым коротким в главе. Единственная проблема в том, можно ли рассматривать 24:00 как время суток — эта запись представляет *исключающий* конец дня в отличие от записи 00:00, представляющей *включающее* начало дня. Значение 24:00 используется не так широко, но иногда его необходимо учитывать.

Но вернемся к более сложным материям и поразмыслим об арифметических операциях с гражданским временем. С машинным временем все просто: всегда можно сложить промежутки, прибавить их или вычесть из момента времени или вычислить разность между двумя моментами для получения промежутка. Все достаточно предсказуемо. Арифметические же операции с гражданским временем могут преподнести немало сюрпризов.

Периоды: арифметические операции с гражданским временем

Тип в `java.time`: `java.time.Period`. Тип в `Noda Time`: `NodaTime.Period`.

Обычно в арифметике существует правильный ответ. Если в начальных классах школы вас спрашивают, сколько будет $5 + 6$, то правильным ответом будет определенно 11. Учитель скажет «правильно» или «неправильно», но никогда не скажет «*может быть*».

С календарной арифметикой дело обстоит иначе (по крайней мере, в граничных случаях, но они встречаются довольно часто, и их нельзя просто проигнорировать). Если вас спрашивают: «Какой день наступит через один месяц после 31 мая 2021 года?», ответы «30 июня 2021 года» и «1 июля 2021 года» выглядят одинаково разумно. Эта неоднозначность представлена на рис. 7.5.

Впрочем, можно определить полезное понятие: *период*. Период напоминает вектор значений для разных календарных единиц: определенного количества лет, месяцев и т. д. Таким образом, *3 года*, *1 месяц* и *2 дня* — период. В библиотеках даты и времени нет единого мнения о том, должны ли периоды останавливаться на днях или переходить к меньшим единицам (например, часы, минуты, секунды и доли секунд).

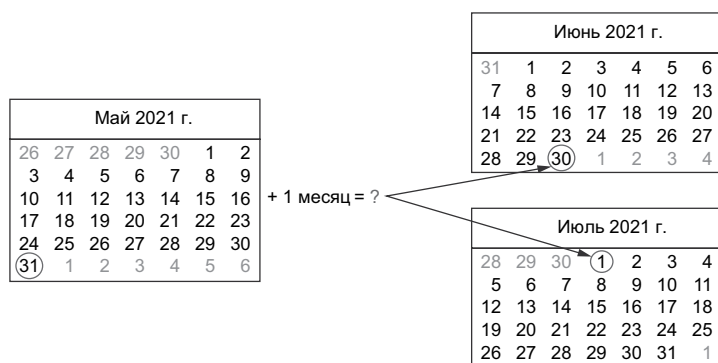


Рис. 7.5. Прибавляя месяц к дате, вы не всегда получите очевидный ответ

ПРИМЕЧАНИЕ

Промежуток всегда представляет фиксированную продолжительность времени независимо от контекста. Три секунды — всегда три секунды. Продолжительность периода может изменяться. Самый очевидный пример — 1 месяц, длительность которого меняется в зависимости от месяца (а в случае февраля — еще и года).

У периодов есть свои странности; например, ничего не мешает создать период из *16 месяцев* или *35 дней*. Хотя *16 месяцев* можно с достаточной уверенностью нормализовать в *1 год и 4 месяца* (если вы работаете в григорианском календаре), *35 дней* определенно не удастся нормализовать в эквивалентный период из *1 месяца и x дней*, потому что при использовании периода значение *x* будет зависеть от конкретного месяца.

Арифметические вычисления только между периодами достаточно прямолинейны: например, если сложить *2 месяца и 3 дня* с *1 годом и 2 днями*, получим *1 год, 2 месяца и 5 дней*. Вопрос о том, всегда ли вычитание имеет смысл, по-разному решается в разных библиотеках; вычитание этих двух периодов с результатом *1 год, 2 месяца и 1 день* нормально сработает на уровне программного кода, но вопрос, будет ли этот период иметь смысл и быть полезным, остается открытым. На более общем уровне вопрос заключается в том, хороши ли периоды со смешанными знаками — с учетом того, что они редко встречаются на практике.

В общем и целом можно назвать следующие типичные операции:

- Дата + Период => Дата
- Дата - Период => Дата
- Дата - Дата => Период (возможно, с указанием используемых единиц)

- Период + Период => Период
- Период - Период => Период

Хотя арифметические операции, выполняемые только с периодами, просты, при введении операций с датой и периодом (первые две операции в приведенном списке) возникают две возможные проблемы. Если вернуться к граничным случаям, упомянутым выше, библиотеки будут давать разные результаты для некоторых вычислений; да и *люди* тоже могут отвечать по-своему. Дело не в том, что в библиотеках ошибка (хотя и такая возможность всегда есть); просто очевидного правильного ответа не существует. Но как бы библиотека ни ответила на ваш вопрос, не исключено, что она нарушит ваши простые ожидания. В частности, многих удивят два аспекта.

Во-первых, сложение в календарных арифметических операциях не *ассоциативно*. Допустим, вы хотите сложить значения *31 января 2021 года*, *1 месяц* и *2 месяца*. Есть два варианта группировки операций:

- (31 января 2021 года + 1 месяц) + 2 месяца
- 31 января 2021 года + (1 месяц + 2 месяца)

В `java.time` и `Noda Time` первая операция дает результат 28 апреля 2021 года, а вторая — 30 апреля 2021 года. К этим результатам приводят следующие шаги:

- (31 января 2021 года + 1 месяц) + 2 месяца
 - 31 января 2021 года + 1 месяц => 28 февраля 2021 года
 - 28 февраля 2021 года + 2 месяца => 28 апреля 2021 года
- 31 января 2021 года + (1 месяц + 2 месяца)
 - 1 месяц + 2 месяца => 3 месяца
 - 31 января 2021 года + 3 месяца => 30 апреля 2021 года

Остальные библиотеки могут давать другие результаты, последовательные в одних случаях, но непоследовательные в других.

Во-вторых, сложение даты и периода необратимо. Иначе говоря, для даты *d* и периода *p* можно ожидать, что результат (*d* + *p*) - *p* всегда будет равен *d*, но это не так. Например, какие бы правила ни использовались в библиотеке, если прибавить месяц к 31 января, а затем вычесть месяц из результата, вы не вернетесь к 31 января.

Если вам кажется, что в реальном мире все это неважно, рассмотрите гипотетическую ситуацию, представленную на рис. 7.6: 28 февраля 2022 года проходят выборы. Избиратели, которым исполняется 18 лет в день выборов, имеют право в них участвовать. Нужно ли разрешить участие избирателям, родившимся 29 февраля 2004 года?

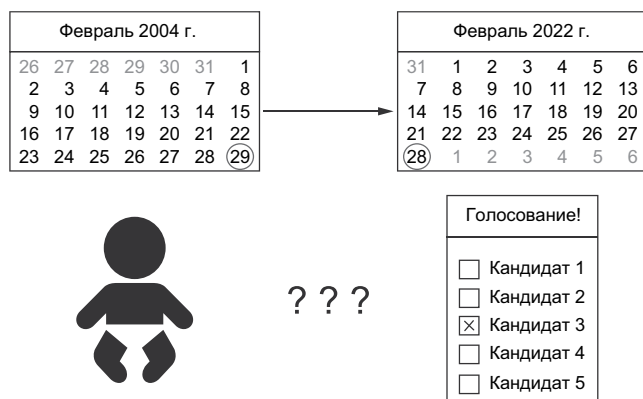


Рис. 7.6. Реальные последствия решений в календарных арифметических операциях

Хотя это был гипотетический пример, такие ситуации происходят. Например, в Великобритании всеобщие выборы проводились 28 февраля 1974 года. По возможности избегайте подобных неоднозначностей, если вы контролируете эти даты.

Как выразить это требование в арифметических операциях? Следующие два варианта *кажутся* разумными:

- Вычесть 18 лет из дня выборов. Каждый, кто уже родился в этот день, может участвовать в выборах.
- Прибавить 18 лет к дате рождения. Избиратель может голосовать, если выборы проходят в этот день или позже.

`java.time` и `Noda Time` ведут себя одинаково, но дают разные результаты для обоих случаев. В первом варианте предполагается, что избиратель голосовать не сможет, потому что 28 февраля 2004 года он еще не родился. Во втором считается, что избиратель *может* проголосовать, потому что прибавление 18 лет к 29 февраля 2004 года вернет 28 февраля 2022 года. Другая библиотека может решить «перенести» результат на 1 марта 2022 года.

Лично я *предполагаю*, что здесь правилен второй вариант, а результат `java.time` и `Noda Time`, вероятнее всего, будет юридически корректным. Но я не уверен, что все страны мира формулируют свои законы именно таким образом, и нельзя исключать, что в каких-то странах существуют неоднозначные или непоследовательные законы.

Я не пытаюсь вас запугать. Скорее призываю хорошенько подумать каждый раз, когда вы выполняете календарные арифметические операции, и убедиться, что ожидания всех вовлеченных сторон одинаковы.

До сих пор мы рассматривали машинное время в контексте четко определенных моментов и гражданского времени, разделяющего ось времени на годы, месяцы и дни, но мы еще не пытались связать машинное время с гражданским. Для этого нам понадобятся часовые пояса.

7.1.3. Часовые пояса, UTC и смещения от UTC

Типы в `java.time`: `java.time.ZoneId` и `java.time.ZoneOffset`. Типы в `Noda Time`: `NodaTime.DateTimeZone` и `NodaTime.Offset`.

Скорее всего, вы уже знаете, что такое часовые пояса — хотя бы в общих чертах. К сожалению, с ними связан ряд распространенных заблуждений, которые я попытаюсь здесь опровергнуть. В этом разделе я буду считать, что григорианский календарь — *единственная* календарная система. На самом деле расширить описание часовых поясов на другие системы нетрудно, но это сильно усложнит объяснения.

Обычно люди представляют себе время так, что полдень наступает примерно тогда, когда солнце находится прямо над головой. В разных точках мира это происходит в разное время. Часовые пояса стараются учесть это обстоятельство. Например, я пишу эти строки в 15:53 в Великобритании. Я знаю, что для жителей Сан-Франциско сейчас 7:53, а для жителей Индии — 21:23.

Часовой пояс, по сути, содержит три блока данных:

- идентификатор или название;
- часть земной поверхности, которая, как считается, находится в этом часовом поясе;
- функцию, связывающую любой момент времени с гражданской датой и временем.

Если представить, что все носят точные наручные часы с правильной настройкой часового пояса, в котором они находятся, то все, кто живет в одном часовом поясе, будут видеть одну дату и время в любой момент времени, поскольку относятся к конкретному часовому поясу. Два человека в разных часовых поясах *могут* видеть одинаковую дату и время. Но даже если прямо сейчас эти значения совпадают, через минуту они могут разойтись.

Когда мы вводили концепцию начала эпохи, я упоминал о том, что начало эпохи Unix было моментом времени, соответствующим полуночи 1 января 1970 года UTC. Что такое UTC? Это нулевой часовой пояс; базовая линия, используемая для описания других часовых поясов. Строго говоря, это вообще не часовой пояс (потому что на земле нет области, которая бы относилась к часовому поясу UTC), но его часто используют в качестве часового пояса — самого

простого из возможных. Я говорю здесь об UTC, потому что это своего рода промежуточный шаг для работы с реальными, более сложными часовыми поясами.

Соотнести момент времени с гражданской датой и временем с использованием UTC несложно. Вы уже знаете дату и время UTC, представленную началом эпохи (1 января 1970 года 00:00:00), и момент времени — всего лишь результат прибавления промежутка к началу эпохи. В UTC каждый день состоит из 24 часов, каждый час — из 60 минут и т. д. Здесь нет раздражающих особенностей других часовых поясов, о которых вы вскоре узнаете. Вам все равно придется иметь дело с високосными годами, но это не так трудно. Моменты времени до начала эпохи также работают достаточно очевидно; например, если вы используете начало эпохи Unix, а момент представлен промежутком –10 секунд, то он соответствует 31 декабря 1969 года 23:59:50.

Теперь, когда вы поняли в общих чертах, что такое UTC, мы можем рассматривать функцию, связывающую любой момент времени с гражданской датой и временем в часовом поясе, в качестве эквивалента функции, связывающей любой момент со *смещением UTC*. Это значение сообщает, насколько опережает или отстает от UTC данный часовой пояс на текущий момент.

Рассмотрим конкретный пример. В момент времени, соответствующий 20 ноября 2020 года 15:53 UTC, смещение UTC для часового пояса Сан-Франциско составляет –8 часов. Считается, что Сан-Франциско на 8 часов отстает от UTC, поэтому там 7:53. В Индии смещение UTC в этот момент составляет 5 часов и 30 минут, а значит, там 21:23.

Но эта функция соотнесения момента времени со смещением UTC не обязана выдавать одинаковые результаты для всех моментов, и в большинстве часовых поясов она этого не делает. Таким образом, например, на 20 июня 2020 года в 15:53 смещение UTC для часового пояса Сан-Франциско составляет –7 часов, и по местному времени будет 8:53. При этом смещение UTC для Индии по-прежнему составляет 5 часов 30 минут — оно постоянно используется с 1945 года.

Хотя соотнесение момента времени с гражданской датой и временем выполняется недвусмысленно, об обратном преобразовании этого сказать нельзя. Некоторые значения гражданской даты и времени *неоднозначны* (с конкретной гражданской датой и временем соотносятся сразу несколько моментов), а некоторые *пропускаются* (не существует ни одного момента, соотносящегося с гражданской датой и временем). Например, в часовом поясе Сан-Франциско смещение поменялось с UTC-7 на UTC-8 1 ноября 2020 года в 2 часа ночи по местному времени (9:00 UTC), когда произошел *возврат с летнего времени*. Это означает, что для любого жителя Сан-Франциско с точными часами четкая последовательность моментов времени будет выглядеть так:

- 01:59:58
- 01:59:59

- 01:00:00 ← здесь происходит *возврат с летнего времени*
- 01:00:01
- 01:00:02

Таким образом, 1 ноября 2020 года гражданское время 1:45 наступает дважды. Два жителя Сан-Франциско смогут сказать, что кошка разбудила их в 1:45, тогда как на самом деле они проснулись с интервалом в 1 час.

С другой стороны, 8 марта 2020 года в Сан-Франциско время смещается на час *вперед* в момент, соответствующий 2 часам по местному времени (10:00 UTC); происходит переход с UTC-8 на UTC-7. Таким образом, в этот день для жителей Сан-Франциско последовательность выглядит так:

- 01:59:58
- 01:59:59
- 03:00:00 ← здесь происходит *переход на летнее время*
- 03:00:01
- 03:00:02

Это означает, что гражданская дата и время 8 марта 2020 года 2:45 вообще не наступает. Любой житель Сан-Франциско, утверждающий, что кошка разбудила его в 2:45 этой ночью, что-то путает.

На рис. 7.7 представлена диаграмма смещений UTC для четырех часовых поясов (Европа/Москва, Европа/Париж, Америка/Асунсьон и Америка/Лос-Анджелес) в 2020 году. Часовой пояс Америка/Асунсьон действует в Парагвае, а пояс Америка/Лос-Анджелес действует в Сан-Франциско. Следует заметить, что Парагвай находится в Южном полушарии, поэтому переход на летнее время в нем происходит в октябре, а возврат — в марте.

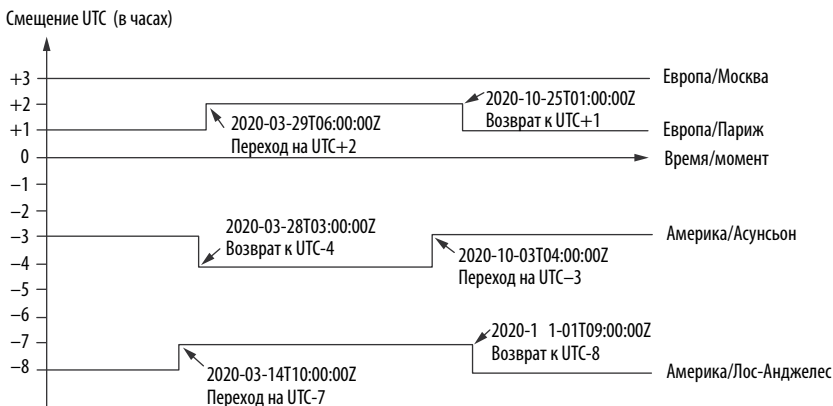


Рис. 7.7. Четыре часовых пояса со смещениями UTC на оси времени

Не обязательно знать во всех подробностях, что и когда происходит в том или ином поясе. (Собственно, для этого и нужны базы данных часовых поясов. Мы вернемся к этой теме позже.) Но нужно помнить, что преобразование момента в гражданскую дату и время в конкретном часовом поясе выполняется однозначно, а при преобразовании в другом направлении существуют граничные случаи, которые необходимо учитывать.

Что не является часовым поясом?

В приведенных выше примерах я намеренно не использовал термины «Тихоокеанское стандартное время» или «Тихоокеанское летнее время» для часового пояса Сан-Франциско. Хотя эти термины часто рассматриваются как своего рода альтернатива для смещения UTC, они не являются часовыми поясами как таковыми. Точнее сказать, что часовой пояс, включающий Сан-Франциско, переключается между Тихоокеанским стандартным временем и Тихоокеанским летним временем. Другие часовые пояса также могут иногда использовать Тихоокеанское стандартное время, а иногда использовать время, отличное от Сан-Франциско. Таким образом, Тихоокеанское стандартное время и другие аналогичные описания в общем случае *не являются* названиями часовых поясов.

ПРИМЕЧАНИЕ

К сожалению, в базе данных часовых поясов Windows название «Тихоокеанское стандартное время» *используется* для обозначения часового пояса, включающего Сан-Франциско; аналогичные обозначения применяются и для многих других часовых поясов. Таким образом, можно запросить описание времени по Тихоокеанскому стандартному времени и получить результат в Тихоокеанском летнем времени. Вместо базы данных часовых поясов Windows я рекомендую использовать часовые пояса IANA, описанные далее.

Если учесть, что описания *неполных часовых поясов* в действительности не являются названиями часовых поясов, вполне логично, что произведенные от них сокращения (такие, как *PST* и *PDT*) тоже не являются такими названиями. Впрочем, сокращения еще хуже описаний, потому что они с большей вероятностью могут оказаться неоднозначными. Совершенно ужасный пример: *BST* является сокращением как от *British Summer Time* (*Британское летнее время*), так и от *British Standard Time* (*Британское стандартное время*), причем последнее обозначение использовалось между 1968 и 1971 годом. Сокращения удобно выводить для пользователей, но для любых других целей их лучше не использовать.

Наконец, смещения UTC сами по себе не являются часовыми поясами. К сожалению, даже ISO-8601 (стандарт текстовых представлений даты и времени) содержит ошибочную трактовку. Значение, описанное как *обозначение зоны* в ISO-8601, представляет только смещение UTC. Это важно, потому что смещение UTC в один момент времени мало что говорит о смещении UTC в другой момент времени в том же часовом поясе. И снова смещения UTC могут быть

очень полезными и более простыми, чем попытки описать реальный часовой пояс, но важно различать эти два понятия.

Например, рассмотрим дату/время и смещение `2021-06-19T14:00:00-04` — иначе говоря, 19 июня 2021 года в 14:00 местного времени в часовом поясе, на данный момент отстающем на 4 часа от UTC. Каким будет смещение UTC 19 декабря в том же местном времени? В Нью-Йорке оно будет равно `-5`; в Асунсьоне (столица Парагвая) оно будет равно `-3`, хотя в обоих местах в июне смещение UTC равно `-4`. Исходная информация *содержит* смещение UTC, но это *не указывает* на часовой пояс.

Откуда берется информация о часовых поясах?

Тип в `java.time`: `java.time.zone.ZoneRuleProvider`. Типы в `Noda Time`: `NodaTime.DateTimeZoneProviders` и `NodaTime.IDateTimeZoneProvider`.

В примечании выше упоминается *база данных часовых поясов Windows*, установленная на всех компьютерах Windows и изменяемая при обновлениях Windows. Тем не менее это не самый популярный источник информации о часовых поясах. Вместо нее почти во всех системах, не входящих в семейство Windows, используется функционирующая на добровольной основе база данных под управлением IANA (Internet Assigned Numbers Authority — Администрация адресного пространства интернета)¹. Из-за долгой истории существования она известна под рядом других названий. Возможно, вы слышали о часовых поясах Olson, zoneinfo, tz или tzdb. Все они относятся к одному источнику данных; новые названия появляются и исчезают со временем.

ПРИМЕЧАНИЕ

У каждой платформы разработки свой подход к получению данных о часовых поясах. Например, в Java по умолчанию используются часовые пояса IANA даже при выполнении в системе Windows. .NET работает с платформенными часовыми поясами, поэтому при выполнении в Linux используются часовые пояса IANA, а в Windows — часовые пояса Windows. В .NET 6 были внесены доработки. Поинтересуйтесь, какая информация о часовых поясах будет использоваться в вашем коде с учетом всех операционных систем, в которых он будет выполняться.

Часовые пояса IANA «обычно идентифицируются по названию континента или океана, а затем по названию самого большого города в регионе» (<https://data.iana.org/time-zones/tz-link.html>).

Часовые пояса, которые до сих пор использовались в примерах:

- Сан-Франциско: Америка/Лос-Анджелес;
- Москва: Европа/Москва;

¹ База данных часовых поясов доступна по адресу <https://www.iana.org/time-zones>.

- Парагвай: Америка/Асунсьон;
- Великобритания: Европа/Лондон;
- Индия: Азия/Калькутта.

Правила часовых поясов меняются несколько раз в год. Здесь я не имею в виду переход часового пояса Америка/Лос-Анджелес с UTC-8 на UTC-7 и обратно; я говорю об изменении в правилах, управляющих этими переходами. Например, Закон об энергетической политике 2005 года поменял правила соблюдения летнего времени в Соединенных Штатах, которые вступили в силу в 2007 году. Правила часовых поясов — предмет политики и определяются государством. Когда группа добровольцев базы данных IANA узнает о пересмотре правил (с однозначным подтверждением того, что изменения действительно были ратифицированы правительством, а не просто предложены), в базу данных вносятся нововведения и она публикуется. Иногда несколько изменений объединяют в один выпуск. Названия выпусков содержат год выпуска и буквенный суффикс (например, в 2020 году первым был выпуск 2020a, за ним последовал выпуск 2020b и т. д.).

Способ передачи этой информации на компьютер, где выполняется программа, сильно зависит от среды. Мы вернемся к этому позже, в разделе 7.4.4, когда будем выяснять возможные последствия этого для кода.

Итак, мы рассмотрели три группы понятий:

- *Машинное время* — моменты времени, начало эпохи и промежутки.
- *Гражданское время* — календарные системы, даты, периоды и время суток.
- *Часовые пояса* — UTC и смещения UTC.

На их основе можно вывести другие понятия, и хорошие библиотеки даты и времени часто предоставляют широкий диапазон типов, при помощи которых код может ясно и точно выразить нужный смысл. Но прежде чем переходить к анализу кода, я кратко коснусь моментов, которые мы не затронули в приведенных выше описаниях.

7.1.4. Концепции даты и времени, вызывающие приступ головной боли

В технических книгах я стараюсь быть точен. Стоит ли читать такую литературу, если информация в ней приблизительная? Но иногда стремление к абсолютной точности противоречит пользе. Я уже упоминал пару аспектов работы с датой и временем, которые не буду детально рассматривать в этой главе, но сейчас расскажу о них подробнее. Этот раздел можно полностью пропустить — он никак не повлияет на дальнейший материал. Но было бы неплохо после тяжелого дня

немного утешить себя мыслью: «Конечно, мне приходится работать с изменением правил часовых поясов, но по крайней мере не нужно возиться с проблемами относительности». Так мы подходим к первой теме...

Относительность

В одном из моих любимых эпизодов сериала «Доктор Кто» доктор говорит: «Люди полагают, что время — это четкая последовательность причин и следствий, но с нелинейной объективной точки зрения оно больше похоже на огромный шар колеблющегося, волнующегося... вещества». Эта формулировка хорошо описывает мой уровень понимания теории относительности. Я понимаю ее достаточно, чтобы ее опасаться, особенно концепции того, что мы (люди и компьютеры) воспринимаем время по-разному в зависимости от системы координат, скорости и ускорения.

Мы начали с определения понятия момента времени как чего-то, что мы все представляем одинаково. Два человека в разных часовых поясах и календарных системах одинаково ответят, что такое *сейчас*. Теория относительности предпологает, что все не так просто, и возможно, даже концепция «*сейчас*» не имеет особого смысла.

Некоторым инфраструктурам (например, GPS) приходится учитывать этот факт. К счастью, в коде бизнес-приложений этого делать не нужно.

Корректировочные секунды

Время — не единственное, что колеблется и волнуется. Вращение Земли тоже не идеально, оно понемногу замедляется (хотя и очень медленно). Это значит, что между «наблюдаемым солнечным временем» (по которому Солнце в полдень на гринвичском меридиане находится прямо над головой) и временем, которое сообщают атомные часы, существуют небольшие расхождения. Корректировочные секунды вводятся для учета этого обстоятельства. Они добавляются (или удаляются, по крайней мере, теоретически) в ось времени UTC для согласования UTC и наблюдаемого солнечного времени.

Корректировочные секунды добавляются или удаляются только для изменения длины последней минуты в конце июня или декабря. Это значит, что, хотя минута обычно состоит из 60 секунд, она может состоять из 61 или 59 секунд. Например, корректировочная секунда, добавленная в конце 2016 года, была вставлена 31 декабря 2016 года в 23:59:60. На момент написания книги *отрицательных корректировочных секунд* еще не было (когда секунда удаляется с оси времени, а не добавляется на нее), но они возможны.

Системы по-разному выводят информацию о корректировочных секундах либо просто делают вид, что их не существует. Например, в некоторых системах используется *размытая корректировка*, при которой лишняя секунда фактически

распределяется по более длинному периоду времени. Таким образом, вблизи от момента вставки корректировочной секунды продолжительность секунды может немного превышать одну секунду. Да, я понимаю, как странно это звучит.

Если всего этого недостаточно, чтобы у вас заболела голова: корректировочные секунды непредсказуемы. Они объявляются заранее, что значительно лучше некоторых изменений часовых поясов, но даже в этом случае это означает, что вам придется тщательно продумывать корректность данных, которые вы будете хранить в будущем. Проблема более подробно рассматривается в разделе 7.4.4. Как и прежде, некоторые инфраструктуры (например, NTP) должны учитывать корректировочные секунды, но в большинстве других программ этого делать не нужно.

Который час на Марсе?

Если вам кажется, что трудно организовать встречу с участниками из разных часовых поясов на Земле, представьте, что на ней должен присутствовать участник с Марса (где продолжительность дня составляет 24 часа 37 минут), с Юпитера (где день немного короче 10 часов) и с Венеры (где продолжительность дня составляет 5832 часа, что длиннее венерианского *года*). Представьте, что когда вы все-таки устроили встречу, в конце кто-то говорит: «Завтра в то же время?»

Идея о том, что новые библиотеки даты и времени должны поддерживать вневременное время, предлагалась вполне серьезно. Надеюсь, к тому моменту, когда она станет важной для широкой коммерческой разработки, я уже буду на пенсии.

Смена календарных систем

В 1582 году в Риме за 4 октября последовало 15 октября. В 1572 году в Лондоне за 2 сентября последовало 14 сентября. Это примеры перехода с юлианского календаря на григорианский — события, которое происходило в разных местах с различными датами.

Это значит, что жители разных стран, которые *обычно* используют одну календарную систему, могут не сойтись во мнениях по поводу дат. Например, сражение при Лоустофте состоялось 13 июня 1665 года... или 3 июня 1665 года, в зависимости от того, на чьей стороне были вы.

Одна из странностей, заслуживающих внимания, — переход Швеции с юлианского календаря на григорианский. Швеция планировала сделать это постепенно, пропуская все високосные дни с 1700 года, пока не произойдет совмещение с григорианским календарем. К сожалению, хотя в 1700 году все прошло по плану, Швеция отвлеклась на Северную войну (1700–1721) и забыла о нем. 1704 и 1708 годы рассматривались как високосные; это противоречило плану, от которого после этого отказались. Чтобы вернуться к юлианскому календарю, Швеция включила в 1712 год *два* корректировочных дня: 29 февраля и 30 февраля.

Некоторые библиотеки даты и времени пытались обрабатывать подобные переходы, хотя я не уверен, моделирует ли хоть одна популярная библиотека шведскую историю. Они приводят к еще более запутанной арифметике, чем обычно, и, на мой взгляд, в общем случае таких нюансов лучше избегать.

Существуют еще более странные граничные случаи, о которых вам почти наверняка не придется беспокоиться. В следующем разделе рассматриваются аспекты, которые определенно *стоит* учесть при планировании функциональности — задолго до того, как вы начнете писать для нее код.

7.2. ПОДГОТОВКА К РАБОТЕ С ИНФОРМАЦИЕЙ О ДАТЕ И ВРЕМЕНИ

Если вы дочитали предыдущий раздел и с нетерпением ожидаете кода, у меня для вас плохие новости: в этом разделе кода тоже практически нет. Обещаю, дойдет и до него, но структура этой главы создавалась как отражение эффективного подхода к работе с датой и временем: если вы тщательно подготовитесь и продумаете подходы заранее, то написать код будет уже несложно. Теперь, когда мы владеем общими понятиями и рабочей терминологией, подумаем, как применить их в реальных продуктах.

7.2.1. Ограничение объема работ

Вы уже видели, что мир даты и времени может быть невероятно сложным. К счастью, в обычных приложениях эта сложность, скорее всего, не понадобится. При планировании целого приложения или отдельной функциональности, в которых используется дата и время, постарайтесь ограничить объем работ и документируйте принятые решения.

Вероятно, начать стоит с исключения самых сложных и нишевых аспектов:

- Нужно ли приложению учитывать эффект относительности?
- Нужно ли учитывать корректировочные секунды?
- Нужно ли работать с датами, находящимися достаточно далеко в прошлом, чтобы пришлось учитывать исторические смены календарных систем?

Если ответ хотя бы на один из этих вопросов положительный, возможно, набор библиотек, которые вы сможете использовать, будет ограничен, а вам *определенно* стоит действовать еще осторожнее, чем обычно, и серьезно исследовать предметную область, для которой пишется код. Я не дам более конкретных советов, так как мне еще не доводилось работать с такими приложениями, но я предполагаю, что в этом случае выбор подходящих типов для представления концепций продуктов будет еще важнее, чем обычно.

Второй уровень сложности связан с календарными системами и часовыми поясами. Потребуется ли работать с другими календарными системами, кроме григорианского календаря? Большинство бизнес-приложений, вероятно, им и ограничится, но, безусловно, найдутся и контрпримеры, особенно если приложение предназначено для религиозного сообщества, в котором важен конкретный календарь. В пользовательских приложениях необходимость поддержки календарных систем, которые предпочитают клиенты, возникает чуть чаще, но прежде чем браться за нее, стоит взвесить пользу и затраты на ее реализацию. (Преимущества зависят от конкретного приложения, а на затраты влияет выбранная технология; уровень поддержки негригорианских календарных систем сильно различается.)

Уровень сложности, связанной с часовыми поясами, может значительно изменяться. Вот лишь некоторые вопросы, которые стоит себе задать:

- Нужна ли вообще поддержка часовых поясов в продукте? Иногда все приложение можно построить на концепциях машинного времени, что сильно упрощает работу.
- Нужно ли продукту взаимодействовать с часовыми зонами, определяемыми другой системой? Если да, то какую базу данных часовых поясов она использует?
- Разрешить ли пользователю выбирать часовые зоны или просто рассчитывать на успешное определение часового пояса по умолчанию?
- Должен ли продукт работать более чем в одном часовом поясе? Если нет, уверены ли вы, что ситуация не изменится?
- Должен ли продукт идеально соответствовать всем изменениям правил часовых поясов, активно отслеживая любые изменения, или же он может просто использовать правила часового пояса, предоставляемые по умолчанию платформой или библиотекой?
- Должен ли продукт хранить данные, которые естественным образом включают информацию о часовом поясе, или взаимодействие с часовым поясом ограничивается выводом информации о нем на экран?
- Насколько серьезное внимание следует уделять переходам часовых поясов в отношении пропущенного и неоднозначного времени? Например, если вы пишете систему для управления школьным расписанием, вряд ли у учеников будут запланированы уроки на момент перехода.

Многим приложениям, которые должны сообщать пользователю дату и время, так или иначе необходимо поддерживать часовые пояса. Тем не менее, отказавшись от встраивания гибкости, которая вам не понадобится, вы заметно упростите себе жизнь. Конечно, здесь тоже возникает компромисс: если вы написали код с утверждением, что вам придется работать только (допустим) с часовым поясом Парижа, то отменить последствия этого решения будет достаточно

трудно. Тем не менее это может значительно повлиять на простоту решения. Один из способов снизить будущие риски — проследить, чтобы все члены команды знали об этих допущениях и хорошо осознавали, когда они опираются на них. Документирование, в каких местах системы допущения действительно важны, упростит возможный откат изменений в будущем.

Обычно управление объемом работ возможно при наличии подробных требований к продукту или функциональности. Продукты относительно редко изменяются так, что в них, например, *неожиданно* появляется необходимость поддержки разных календарных систем. (Хотя это, конечно, возможно. Такие новые требования скорее становятся частью выхода на новые рынки, чем частью добавления новой функциональности.) Разработчикам, скорее всего, стоит обсудить вопросы между собой, а затем документировать и сверить результаты с владельцами продукта.

ПРИМЕЧАНИЕ

Под термином «*владельцы продукта*» я понимаю людей, которые отвечают за принятие решений о том, что должен делать продукт. В разных компаниях могут использоваться другие названия — например, *продакт-менеджеры*. В зависимости от конкретной модели разработки это могут быть люди из той же компании, что и разработчики, и/или другой компании. Это могут быть и сами разработчики, но эту роль стоит рассматривать отдельно от принятия решений о реализации.

Впрочем, когда дело доходит до подробных требований, владельцы продукта должны участвовать в их проработке.

7.2.2. Уточнение требований к дате и времени

Начну этот раздел с предупреждения: если вы начнете проверять, что требования к продукту, относящиеся к дате и времени, ясны и однозначны, это вряд ли прибавит вам популярности. Скорее всего, очень многие ответят вам: «Разве это не очевидно?» — даже если то, что очевидно для одного человека, отличается от того, что совершенно ясно для других. Но усилия не пропадут даром. Когда требования становятся понятными, программирование упрощается. Без четко сформулированных требований может оказаться, что у людей, участвующих в разработке продукта, разные ожидания, что приводит к хаосу.

Конечно, вы сами решаете, как планировать и документировать требования. Не существует одной обязательной методологии. Вы можете спроектировать обширную начальную архитектуру или проектировать отдельные функции при переходе на более гибкую разработку. Впрочем, если вы работаете по принципу «*проектируйте только то, что нужно прямо сейчас*», будьте аккуратны; если на первом спринте для представления информации вам нужна только дата, но к четвертому окажется, что нужны дата и время (а возможно, и часовой пояс), это значительно усложнит вам жизнь. Старайтесь по возможности предвидеть

будущие естественные требования, но не слишком углубляйтесь во все возможные варианты.

В широком смысле существуют два типа решений, которые необходимо фиксировать в документации: о способе интерпретирования каждого фрагмента данных, относящихся к дате и времени, и о подходе к их обработке. Также необходимо рассмотреть представления хранения и передачи данных, но они скорее относятся к подробностям реализации, чем к требованиям продукта. Два этих вида решений взаимосвязаны, но мы будем рассматривать их по отдельности.

Чтобы описание было конкретным, воспользуемся сценарием с интернет-магазином. Вкратце рассматриваемые требования показаны на рис. 7.8: *покупатели могут вернуть товары в течение 3 месяцев*. К концу сценария у нас появится набор требований, которые можно реализовать и тестировать.

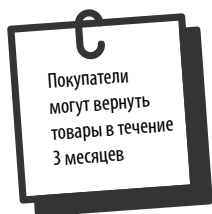


Рис. 7.8. Высокоуровневое требование, которое нуждается в детализации

Выбор подходящих концепций или типов данных

В хороших требованиях к продукту обычно указано, какие данные собираются в конкретной ситуации (и возможно, какие данные намеренно игнорируются). Иногда эта информация выражена неявно и частично скрыта в описании пользовательского взаимодействия, но при явном выражении она более понятна. Информацию, относящуюся к дате и времени, обычно легко заметить, но может быть сложнее решить, как ее обрабатывать.

Первое практическое правило: рассмотрите источник данных. Если вы сохраняете данные о том, что «что-то произошло», обычно вы начинаете с *момента*, в который произошло событие. *Также* можно сохранить часовой пояс (или в более общем случае — место), если он будет актуален для других операций. Сохранение момента обычно выполняется тривиально — в большинстве баз данных и системах ведения журналов реализованы встроенные метки времени.

ПРИМЕЧАНИЕ

Возможно, также стоит выбрать, какой источник текущего времени важнее: если вы сохраняете *текущий момент* как в базе данных, так и на отдельном веб-сервере, синхронизация двух этих часов может быть не идеальной. Насколько это важно, зависит от приложения.

Если вы сохраняете значение даты и времени, предоставленное пользователем, ситуация меняется. Вы используете гражданское, а не машинное время, даже если пользователь сообщает о произошедшем событии. Вам почти наверняка потребуются учитывать информацию часового пояса или, по крайней мере, смещение UTC. Возможно, у вас возникнет соблазн преобразовать ее в момент времени, но я рекомендую точно сохранить информацию, полученную от пользователя, или хотя бы разобранное, но не обязательно преобразованное представление. Когда мы будем рассматривать некоторые граничные случаи, вы увидите, в каких из них подход с *сохранением только времени UTC* может оказаться неподходящим, особенно при регистрации информации о будущем.

Понятно, что для требования о возврате товаров необходимо сохранить некоторые данные, но не очевидно, что это за данные, не говоря уже об используемом представлении. Первый вопрос, который следует задать владельцу продукта: «Покупатели могут вернуть товары в течение 3 месяцев... от *чего?*» Возможные варианты ответа:

- в течение 3 месяцев от нажатия пользователем кнопки Pay (Оплатить);
- в течение 3 месяцев от принятия платежа;
- в течение 3 месяцев от подтверждения заказа;
- в течение 3 месяцев от резервирования товара;
- в течение 3 месяцев от отправки заказа;
- в течение 3 месяцев от получения заказа.

О том, что означает *3 месяца*, мы подумаем позже, но даже в приведенном списке представлены шесть разных точек отсчета. В пятом пункте (*с отправкой*) можно даже выделить еще несколько моментов, но для простоты будем считать, что мы успешно согласовали один из них.

Важно, что все они *являются* моментами времени, и имеет смысл сохранить все эти данные для заказа. Некоторые аспекты могут действовать на уровне отдельных позиций, а не заказа в целом; например, это относится к резервированию товаров и даже к отправке — заказ может быть отправлен в нескольких посылках. Владелец продукта должен учесть все эти аспекты в контексте *возможности возврата товаров в течение 3 месяцев*.

Предположим, владелец продукта отвечает, что для любого конкретного товара покупатель может вернуть этот товар в течение 3 месяцев от момента отправки. (Таким образом, окно возврата может зависеть от товара даже в одном заказе.) Замечательно — требования стали намного точнее.

Скорее всего, вы будете регистрировать и другие моменты времени, но мы знаем, что необходимо сохранить момент отправки каждого товара. Впрочем, это еще не окончательное решение, и здесь можно обратиться к ранее рассмотренным

понятиям и задать еще несколько вопросов. Мы знаем, что 3 месяца — период, а не промежуток, а прибавить период к моменту времени нельзя. Необходимо вывести из момента времени другие данные, чтобы рассматривать его в гражданском времени. Это означает, что нам *придется* учесть календарные системы и часовые пояса.

ПРИМЕЧАНИЕ

Все мы знаем, что требования к продукту могут меняться. Решение о том, что *время отправки определяет окно возврата*, может измениться, как и решения, которые будут приниматься позже. Хранение всех необработанных и независимых от реализации данных с самого начала позволит изменять решение на более поздней стадии. А это означает, что следует регистрировать *все* моменты, перечисленные ранее, и сохранять их как моменты, даже если затем из них будет извлекаться дополнительная информация.

Эта рекомендация связана с предыдущим советом относительно сохранения *данных, полученных от пользователя*. Эти данные могут быть важны, если пользователь задал дату и/или время. Независимой информацией в этом случае является не момент, сохраненный машинными часами, а пользовательский ввод.

Сначала можно спросить владельца продукта, какую календарную систему использовать. Скорее всего, ответ будет простым: григорианскую календарную систему, независимо от пользователя. (Если владелец продукта на этой стадии даст любой другой ответ, стоит предусмотреть намного больше времени на тестирование.)

Затем можно спросить владельца продукта, какой часовой пояс его интересует. Именно здесь полезно привести пример, чтобы обсуждение было более конкретным. Можно рассмотреть некоторые сценарии:

- веб-сервер в Бразилии;
- хранение информации в базе данных в Нью-Йорке;
- размещение заказа для компании, базирующейся в Калифорнии;
- отправка товаров со склада в Техасе;
- покупатель с адресом выставления счета в Берлине (Германия);
- отправка по адресу в Сиднее (Австралия).

Момент, в который товар считается *отправленным*, представляет собой разное локальное время, а возможно, даже разные *даты* для каждого из этих мест. Что здесь важно? Подсказка: наверняка не веб-сервер и не база данных. Практически любой другой ответ будет приемлемым, но поведение продуктов почти никогда не зависит от физического местоположения задействованных компьютеров, если только пользователи не сидят перед ними.

Даже если владелец продукта считает, что эта ситуация притянута за уши, он должен выбрать ответ, который кажется ему правильным, и документировать это решение. Оно естественным образом задает отправную точку для приемочных тестов.

Допустим, владелец продукта отвечает, что актуальным часовым поясом должен считаться тот, в который отправляется товар, — в данном случае это Сидней (Австралия). Превосходно. Это, скорее всего, не значит, что нам *придется* сохранять дополнительные данные; у нас уже есть адрес, по которому осуществляется доставка (по нему можно определить часовой пояс), момент отправки товара как независимая точка отсчета и принятое ранее решение о том, что мы *всегда используем григорианский календарь*. Момент можно преобразовать в местное время адреса поставки, когда вы сочтете нужным. *Возможно*, что его удобно сохранить непосредственно в базе данных, но это детали реализации. Имея эту информацию, можно переходить к остальным вопросам о функциональности.

Вопросы о поведении

Общее утверждение о том, что *покупатели могут возвращать товары в течение 3 месяцев*, нуждается в уточнениях. Мы выявили отправную точку для 3 месяцев, но прежде чем переходить к реализации чего-либо, нужно собрать еще много данных. Конечно, любой владелец продукта, ответственно выполняющий свои обязанности, включает в требования многие подробности, но мы сейчас сосредоточимся на тех из них, которые касаются даты и времени.

Допустим, фактический путь пользователя документирован по следующей схеме:

При просмотре завершенного заказа на веб-сайте любой товар, который был отправлен менее 3 месяцев назад, отображается с возможностью возврата. Когда клиент щелкает по ссылке возврата, открывается форма с подробной информацией. После ее заполнения запускается процедура возврата.

С процедурой возврата связано множество нюансов, но сейчас необходимо прояснить два аспекта даты и времени.

Во-первых, к какому моменту применять 3 месяца — когда пользователь просматривает завершенный заказ, когда он кликает по ссылке, запускающей процесс возврата, или когда он нажимает «Отправить» форму возврата? Это три разных момента времени. Представьте, как будет раздражать покупателя, если он просматривает заказ с возможностью вернуть товар, а при переходе по ссылке возврата минутой позже появляется сообщение, что она уже недействительна. С другой стороны, нельзя допускать вариант, когда пользователь оставляет окно браузера открытым на годы и фактически имеет неограниченный период возврата. Тот же вопрос относится и к заполнению формы.

Возможный набор требований с дополнительной детализацией может выглядеть так:

При просмотре завершенного заказа на веб-сайте любой товар, отправленный менее 3 месяцев назад, отображается с возможностью возврата. Когда клиент переходит по ссылке, сервер проверяет, был ли возврат возможен 5 минут назад, и если нет — возвращает ошибку. Это позволяет покупателям ожидать до 5 минут между просмотром заказа и запуском процесса возврата. (Также это означает, что если покупатель ожидал более 5 минут, но еще остается в границах периода возврата, он все равно сможет перейти к форме возврата.) Если проверка проходит, открывается форма с указанием, что ее необходимо заполнить в течение 2 часов.

При отправке формы сервер проверяет, что процедура возврата была начата за последние 2 часа, и возвращает ошибку, если это не так. Если проверка проходит успешно, форма направляется на обработку и на экране выводится подтверждение для покупателя.

В этом варианте используются два вида ограничений времени: один предоставляет период отсрочки 5 минут за пределами жесткого требования «процесс возврата должен быть начат до времени x », а другой ограничивает продолжительность заполнения самой формы возврата.

Мы уже на полпути к грамотному набору требований, относящихся к дате и времени. Впрочем, в утверждении «менее 3 месяцев» осталась еще одна неочевидная проблема. Мы уже решили, что отсчет 3 месяцев должен начинаться с момента отправки заказа и 3 месяца должны ориентироваться на часовой пояс адреса доставки. Однако в отношении точности еще не все ясно.

Как было показано в примере с голосованием, арифметические операции с календарями не подчиняются обычным математическим правилам. Таким образом, в данном случае необходимо понять разницу между «определить время отправки и прибавить к нему 3 месяца» и «определить текущее время и вычесть из него 3 месяца». Владелец продукта также должен определиться с детализацией: если отправка была выполнена в 10 часов утра, должны ли три месяца завершиться именно в 10 часов утра через три месяца? Покупателям такое решение может показаться неочевидным. Конечно, если владелец продукта выберет этот вариант, он превращается в требование. Однако если бы я был владельцем продукта, то описанные мной условия выглядели бы примерно так:

Возможность возврата товара определяется датой его отправки в часовом поясе адреса доставки. Последняя дата, в которой еще может быть выполнен возврат, вычисляется прибавлением 3 месяцев к текущей дате адреса доставки на момент отправки товара. Если прибавление 3 месяцев к дате отправки переходит через конец месяца, то используется начало следующего месяца. (Пример: если товар отправляется 30 ноября, то

последней допустимой датой возврата будет 1 марта, а не последний день февраля.) Ссылка для возврата отображается для пользователя, пока текущая дата в месте доставки не превысит последнюю допустимую дату возврата.

Формулировка получается многословной, но однозначной. В ней определяется:

- уровень детализации (дата возврата, а не дата и время);
- природа арифметических вычислений с календарем (сложение с начальной датой);
- природа проверки (последняя дата *включается*);
- используемый часовой пояс (адрес доставки);
- способ разрешения арифметических операций с календарем (перенос на начало следующего месяца).

Последнее требование может быть не самым простым для реализации — это зависит от используемой библиотеки, но, по крайней мере, оно четко сформулировано и хорошо тестируется.

Я бы не ожидал, что владелец продукта сформулирует подобные требования самостоятельно, если только он уже не работал таким образом с датой и временем. Пока вы не знаете о странностях календарной арифметики, возможные неоднозначности не всегда очевидны. Но команда разработки может проверять требования к дате и времени, пока они не станут достаточно точными. Процесс преобразования смутного набора требований к продукту в конкретный, однозначный тестируемый набор зависит от организации команды, но очень важно, чтобы в итоге вы пришли к желаемому результату. Возможно, для этого придется не раз задать вопросы с нелепыми граничными случаями или же команда разработчиков сможет предложить более конкретную версию неоднозначных требований. На последнем шаге перед написанием кода необходимо убедиться, что вы правильно подобрали инструментарий.

7.2.3. Использование подходящих библиотек или пакетов

И хотя написать понятный, удобочитаемый код с использованием плохих библиотек даты и времени можно, это не так просто. Если у вас есть четкий набор требований, то у вас есть и основания для оценки технологий, используемых для их реализации.

Впрочем, среда меняется со временем. Например, на момент написания книги предложение `Temporal` с новым набором стандартных объектов для работы с датой и временем в JavaScript еще не было принято. Но когда (и если) его примут, этот вариант стоит рассмотреть для новых проектов JavaScript.

Мы с удовольствием представим рекомендации для Java и .NET, так как автор лучше всего знаком с этими платформами и обе они достаточно стабильны в отношении доступных опций. Конечно, с момента написания текста до его прочтения может появиться что-то новое, но в любом случае эти рекомендации станут хорошими отправными точками.

Начнем с платформы Java. Если у вас есть возможность использовать пакет `java.time`, представленный в Java 8, выберите его. Если вы почему-то ограничены Java 6 или Java 7, проект ThreeTenBackport (<https://www.threeten.org/threetenbp/>) будет хорошей альтернативой. Главное — избегайте `java.util.Date` и `java.util.Calendar`; обе библиотеки полны ловушек, из-за которых неопытный разработчик может написать некорректно работающий код.

Для .NET мы *крайне субъективно* рекомендуем использовать Noda Time (<https://nodatime.org>). Конечно, ничто не мешает эффективно пользоваться встроенными типами (`DateTime`, `DateTimeOffset`, `TimeZoneInfo`, `TimeSpan`), но они не делят логические понятия, рассмотренные ранее, по разным типам. Например, не существует типа для представления даты, а для представления понятий промежутка и времени суток используется один и тот же тип. (С выходом .NET 6 ситуация немного изменилась, но не будем углубляться в подробности.) А это значит, что есть риск написать код, который выглядит правильно, но на самом деле выполняет некорректные операции с нормальными данными (например, сложение получаса с датой). Тот факт, что `DateTime` может означать «в неуказанном часовом поясе», «в системном местном часовом поясе» или «в UTC», тоже не упрощает ситуацию.

Но кроме этих конкретных примеров существуют более общие вопросы, по которым можно проверить любую конкретную библиотеку для вашей платформы.

- Если нужно использовать другие календарные системы, кроме григорианской, поддерживаются ли они библиотекой?
- Предоставляет ли библиотека достаточную степень контроля над используемыми данными часового пояса? (Например, если нужно работать с идентификаторами часовых поясов IANA, лучше не выбирать библиотеку, поддерживающую только часовые пояса Windows.)
- Поддерживает ли библиотека все понятия, выявленные в требованиях, и предоставляет ли она достаточные различия между этими понятиями, чтобы четко выразить намерения в коде?
- Предоставляет ли библиотека неизменяемые типы? Хотя у неизменяемости в общем есть четко выраженные плюсы и минусы, как было показано в главе 4, в контексте библиотеки даты и времени она почти всегда дает преимущество.
- Всегда ли внешние зависимости (например, базы данных, другие библиотеки, сетевые API и т. д.) ведут в направлении конкретной библиотеки? Если нужно выполнять преобразования между разными представлениями, насколько легко это делать?

Там, где это возможно, стоит попытаться создать прототипы некоторых требований к дате и времени для рассматриваемой библиотеки, чтобы примерно представлять, как будет выглядеть код. Обычно это делается в небольшом консольном приложении или проекте модульных тестов, изолированном от кода приложения. Например, с требованиями к возврату товаров, описанными выше, я бы написал модульные тесты для проверки логики того, должна ли отображаться ссылка *возврата товара*. Если вы оцениваете несколько библиотек, возможно, вам удастся создать один набор тестовых сценариев, которые затем реализуются с разными библиотеками. А когда у вас появится *работоспособный* код, использующий все библиотеки, сравните реализации на удобочитаемость. Ну и после того, как вы задокументируете требования в масштабе приложения, обсудите с владельцем продукта требования к конкретным функциям и выберете подходящую библиотеку, можете переходить к написанию кода рабочей версии.

7.3. РЕАЛИЗАЦИЯ КОДА ДАТЫ И ВРЕМЕНИ

Даже пройдя всю необходимую подготовку, к написанию кода следует подходить дисциплинированно. Потерять контроль над ситуацией в поисках кратчайшего пути очень легко, а разобраться в возникшей путанице будет непросто.

7.3.1. Последовательное применение концепций

Последовательное применение концепций в приложении поможет предотвратить возможные ошибки. Осуществить это может быть трудно, если некоторые фрагменты данных иначе используются в разных контекстах. Например, политика возврата в нашем сценарии базируется на фактической дате отправки, но та, в свою очередь, базируется на моменте *отгрузки со склада* в сочетании с часовым поясом места, куда отправляется товар. Действовать последовательно можно и в этом случае, но нужно четко обозначить, что имеется в виду, каждый раз, когда в коде приходится иметь дело с *моментом отправки*.

Информация о дате и времени обычно существует в трех разных формах:

- *в памяти при выполнении кода* — обычно в форме объектов из используемой библиотеки даты и времени;
- *в сетевых запросах во время передачи информации между компьютерами* — как правило, в текстовой форме, особенно для веб-приложений; разработчик отвечает за то, что отправитель и получатель используют и ожидают данные в одном формате. Тем не менее это может быть и двоичный протокол — обычно непрозрачный; в этом случае вам не нужно знать (да вас это и не интересует) смысл фактически передаваемых байтов;

- *в системах хранения данных (файлах в формате JSON, CSV или XML, базе данных)* — как и в случае с сетевыми запросами, можно не контролировать точный формат данных. Однако часто тип данных можно выбирать с использованием полей SQL или стандартных представлений в текстовом формате.

И снова очень важна последовательность. Например, если одна часть приложения позволяет пользователю задать дату (без времени), следует позаботиться о том, чтобы поток информации соблюдал этот выбор во избежание дальнейшей путаницы. Это может быть запрос HTTP, содержащий текстовое значение `2020-12-20`, которое затем парсится в `java.time.LocalDate` и сохраняется в поле с типом `DATE` базы данных. Ничто не мешает написать нормально работающее приложение, которое использует разные концепции даты и времени для этих трех уровней, но код получится очень запутанным. Конечно, я выбрал очень простой пример, обычно в жизни все сложнее.

Проблема рассогласования представлений

При использовании подходящей библиотеки даты и времени в основной части приложения нередко выясняется, что база данных не имеет такого богатого набора типов или что в коде интерфейсной части может использоваться отличающийся набор типов. Продолжим приведенный выше пример: предположим, вы передали дату, выбранную пользователем, в текстовом виде и работали с ней в коде в виде `LocalDate`, но затем ее приходится сохранять в базе данных, поддерживающей единственный тип для работы с датой и временем — метку времени. Что делать? Однозначного ответа нет, есть варианты.

Первый вариант — переход в концепцию, поддерживаемую базой данных. В нашем случае можно преобразовать `LocalDate` в момент `Instant`, представляющий *полночь в начале заданной даты в UTC*. Преимущество такого подхода — возможность задействовать другую функциональность даты и времени внутри базы данных. Это решение легко использовать в другом коде. Однако может показаться, что можно создавать моменты времени, не представляющие полночь любой даты в стандарте UTC.

Второй вариант — использование текстового поля. Например, дата может храниться в том виде, в котором она была получена от интерфейсной части, то есть `2020-12-20`. Это более наглядно показывает, что в поле хранится обычная дата, и если при этом используется формат ISO из примера (год-месяц-день), то ее легко сортировать. С другой стороны, ее не столь эффективно хранить в базе данных и сложнее использовать в запросах.

Третий вариант — использование числового поля с четко определенным значением. Например, дата может представляться *количеством дней с 1 января 1970 года*. Такое решение может быть эффективным с точки зрения хранения и запросов

данных, но потребует более сложного кода во всех системах, напрямую работающих с базой данных, а также усложнит понимание данных в инструментах баз данных (например, SQL Server Management Studio).

ПРИМЕЧАНИЕ

Постарайтесь привести входные данные к предпочтительному типу данных в памяти как можно раньше и приведите выходные данные к итоговому типу как можно позже. Тем самым минимизируется объем кода, необходимого для работы с непоследовательным представлением. Кроме того, это одна из областей, в которых важен принцип DRY («*Не повторяйтесь*»); сам код преобразования должен быть централизован, чтобы избежать любых непоследовательностей при выполнении преобразований.

Подобные рассогласования представлений встречаются довольно часто, но это не единственный случай, когда такие усилия необходимы в контексте преобразования концепций на границах систем.

Концепции, связанные с конкретными приложениями

Иногда оказывается, что одна из естественных концепций приложения не вполне соответствует стандартным концепциям, описанным выше или представленным в библиотеке (либо в используемой базе данных). Примером может послужить *финансовый квартал* с информацией, зависящей от конкретной схемы учета, используемой компанией. Подобные нестандартные концепции встречаются довольно редко, но полезно знать о такой вероятности и планировать свои действия в таком случае.

Как и прежде, важна последовательность. При обнаружении новой концепции ее стоит инкапсулировать тем способом, который воспринимается идиоматически для используемой вами библиотеки. Включите любые нужные преобразования, спроектируйте подходящие текстовые представления и проработайте воплощение концепций в системах хранения данных.

Насколько рано все это нужно сделать? Здесь тоже существует компромисс. Чем скорее начать работать над инкапсуляцией новой концепции, тем больше гибкости будет при проектировании и тем меньше хлопот будет с преобразованием существующего кода для использования нового представления. С другой стороны, если принять все решения на основании одного примера использования концепции, возникает риск чрезмерной подгонки архитектуры под этот пример. В результате полученное решение не будет соответствовать требованиям дальнейших сценариев использования. Чтобы свести к минимуму эти риски, стоит начать поиск других примеров, столкнувшись с первым. Не обязательно проектировать все аспекты будущей функциональности, для которых могут потребоваться данные, но стоит хотя бы подумать, какие операции могут понадобиться и какие ограничения будут при этом действовать.

Среди прочего, эффективная инкапсуляция должна способствовать интеграции тестируемости в архитектуру с самого начала. Впрочем, это относится не только к специальным концепциям; о тестировании стоит помнить постоянно в рамках всей кодовой базы.

7.3.2. Отказ от значений по умолчанию в целях улучшения тестируемости

Как уже говорилось при обсуждении политики возврата интернет-магазина, в документах с требованиями полезно привести ряд примеров. Они идеальны для преобразования в модульные тесты, но только если код тестируемый. В некоторых библиотеках обеспечить это не так просто, как хотелось бы, но эти недостатки легко обойти, придерживаясь некоторых правил.

Рассмотрим конкретный пример, в котором внешне простой код содержит ряд скрытых утверждений. (В нем используются классы из пакетов `java.util` и `java.text`; с удовольствием отмечу, что `java.time` решает минимум две из описанных проблем.)

```
String now = DateFormat.getDateInstance().format(new Date());
```

В этой строке кода скрыто несколько решений, поскольку разработчики платформы посчитали, что будет удобно сделать эти решения неявными. Эту строку также сложно тестировать — вероятно, из-за того, что тестируемости не было в списке приоритетов во время проектирования.

- Использование системных часов означает, что нельзя протестировать, что происходит в конкретные моменты времени.
- Текущий момент времени преобразуется в системный часовой пояс, что затрудняет тестирование кода в разных часовых зонах.
- Используется календарная система по умолчанию для локального контекста по умолчанию.
- Используется формат даты локального контекста по умолчанию.

Эти утверждения упрощают жизнь, если тестировать этот код не нужно и если вы пишете настольное приложение, в котором, скорее всего, будет использоваться текущий культурный контекст и часовой пояс. В любой другой ситуации такого кода лучше избегать. Форматированием строк мы займемся позже, а пока рассмотрим три первых аспекта.

Существующие абстракции часов

Современные библиотеки даты и времени часто абстрагируют концепцию часов. Впрочем, даже если они этого не делают, вы можете сделать это самостоятельно. Пакет `java.time` содержит абстрактный класс `Clock`, который предоставляет

часовой пояс, а также сервис *определения текущего момента времени*. Noda Time включает интерфейс `IClock` с единственным методом `GetCurrentInstant()`. В обоих случаях предоставляются возможности получения экземпляров для тестовых целей. В любой ситуации, когда коду нужно узнать текущий момент времени, мы рекомендуем внедрение зависимостей для получения доступа к часам — вместо любых решений, в которых всегда используются системные часы.

Если вам не очевидно, почему это необходимо для тестирования, рассмотрим искусственный простой пример. Допустим, вы хотите создать класс, который может определить, находится ли текущий момент времени на расстоянии в пределах одной минуты от некоторого целевого момента. В реальном коде целевой момент лучше сделать гибким, используя параметр `Duration` при конструировании, но для простоты мы его жестко зафиксируем. В следующем листинге приведен довольно несложный код, использующий системные часы.

Листинг 7.1. Класс `OneMinuteTarget`, непригодный для тестирования

```
public final class OneMinuteTarget {
    private static final Duration ONE_MINUTE = Duration.ofMinutes(1);
    private final Instant minInclusive;
    private final Instant maxInclusive;

    public OneMinuteTarget(@NonNull Instant target) {
        minInclusive = target.minus(ONE_MINUTE);
        maxInclusive = target.plus(ONE_MINUTE);
    }

    public boolean isWithinOneMinuteOfTarget() {
        Instant now = Instant.now();
        return now.compareTo(minInclusive) >= 0 && now.compareTo(maxInclusive) <= 0;
    }
}
```

Эта строка затрудняет тестирование кода

Как тестировать этот код? Я бы протестировал пять сценариев:

1. Текущий момент более чем на минуту предшествует целевому моменту.
2. Текущий момент ровно на одну минуту предшествует целевому моменту.
3. Текущий момент менее чем на минуту предшествует целевому моменту либо следует менее чем через минуту после целевого момента.
4. Текущий момент следует ровно через одну минуту после целевого момента.
5. Текущий момент следует более чем через минуту после целевого момента.

С кодом из листинга 7.1 точно протестировать эти сценарии не удастся. Можно написать код для тестов 1, 3 и 5, сделав обоснованные утверждения относительно скорости выполнения тестов, но нельзя быть уверенным, что системные часы показывают время *ровно* на 1 минуту до или после целевого момента. Мы можем узнать, когда начинается выполнение теста, но мы не знаем, сколько времени

пройдет между этим моментом и вызовом `Instant.now()` в тестируемом методе. Но если внедрить `Clock` в конструктор, как показано в следующем листинге, этот код можно будет тестировать.

Листинг 7.2. Тестируемый аналог листинга 7.1 с использованием `java.time.Clock`

```
public final class OneMinuteTarget {
    private static final Duration ONE_MINUTE = Duration.ofMinutes(1);
    private final Clock clock;
    private final Instant minInclusive;
    private final Instant maxInclusive;
    public OneMinuteTarget(@Nonnull Clock clock, @Nonnull Instant target) {
        this.clock = clock;
        minInclusive = target.minus(ONE_MINUTE);
        maxInclusive = target.plus(ONE_MINUTE);
    }

    public boolean isWithinOneMinuteOfTarget() {
        Instant now = clock.instant();
        return now.compareTo(minInclusive) >= 0 && now.compareTo(maxInclusive) <= 0;
    }
}
```

Часы, к которым мы обращаемся каждый раз, когда требуется получить текущий момент

Получить часы, предоставленные вызывающей стороной, для дальнейшего использования

Статический метод, непригодный для тестирования, заменяется вызовом метода часов

Теперь легко написать тесты для разных ситуаций. Параметризованные тесты часто полезны при работе с датой и временем, как показано в следующем листинге.

Листинг 7.3. Тестирование класса `current-time-sensitive` с использованием `Clock.fixed`

```
class OneMinuteTargetTest {
    @ParameterizedTest
    @ValueSource(ints = {-61, 61})
    void outsideTargetInterval(int secondsFromTargetToClock) {
        Instant target = Instant.ofEpochSecond(10000);
        Clock clock = Clock.fixed(
            target.plusSeconds(secondsFromTargetToClock),
            ZoneOffset.UTC);
        OneMinuteTarget subject = new OneMinuteTarget(clock, target);
        assertFalse(subject.isWithinOneMinuteOfTarget());
    }

    @ParameterizedTest
    @ValueSource(ints = {-60, -30, 60})
    void withinTargetInterval(int secondsFromTargetToClock) {
        Instant target = Instant.ofEpochSecond(10000);
        Clock clock = Clock.fixed(
            target.plusSeconds(secondsFromTargetToClock),
            ZoneOffset.UTC);
        OneMinuteTarget subject = new OneMinuteTarget(clock, target);
        assertTrue(subject.isWithinOneMinuteOfTarget());
    }
}
```

Указание тестируемых значений

Создание произвольного целевого момента

Конструирование часов с временем относительно целевого момента

Здесь используются два метода: для тестирования моментов за пределами целевого интервала и для тестирования моментов в этом интервале. Учитывая, что методы отличаются только параметризованными значениями и вызовом `assertFalse/assertTrue`, можно обойтись и одним методом, который параметризуется по ожидаемому результату. Конкретная структура тестов выходит за рамки этой главы; здесь важно, что возможность управления временем упрощает тестирование кода.

Создание собственной абстракции часов

Если вы используете библиотеку даты и времени, в которой еще нет подходящей абстракции, просто создайте ее сами — в идеале в библиотеке, которую можно применить повторно в любом другом приложении, использующем ту же библиотеку даты и времени. Решайте сами, оставить ли чистую абстракцию *текущего момента* (как в Noda Time) или же включить часовой пояс (как в `java.time`). Обычно код делится на три типа:

- абстрактный класс или интерфейс, от которого зависит большая часть кода;
- реализация с единственным экземпляром, использующая системные часы;
- фиктивная реализация, позволяющая вызывающей стороне задать момент времени при конструировании или позже. Доступ к ней может предоставляться через специальный тестовый пакет, чтобы предотвратить зависимость от нее в окончательной версии кода.

Для конкретики представьте, что `java.time` *не предоставляет* абстракцию часов или что вы хотите использовать абстракцию, ограниченную текущим моментом (без включения часового пояса). Можно задать собственный интерфейс `InstantClock`, как показано в следующем листинге.

Листинг 7.4. Задание интерфейса часов для `Instant`

```
public interface InstantClock {
    Instant getCurrentInstant();
}
```

Далее можно реализовать интерфейс в одиночном экземпляре `SystemInstantClock`.

Листинг 7.5. Реализация `InstantClock` в одиночном экземпляре системных часов

```
public final class SystemInstantClock implements InstantClock {
    private static final SystemInstantClock instance =
        new SystemInstantClock();

    private SystemInstantClock() {}

    public static SystemInstantClock getInstance() {
        return instance;
    }

    public Instant getCurrentInstant() {
        return Instant.now();
    }
}
```

Предотвращает создание других экземпляров

Открытый метод для обращения к одиночному экземпляру

Делегирование метода `Instant.now()`, который использует системные часы

Наконец, можно создать фиктивный объект (fake) для тестовых целей, как показано в следующем листинге.

Листинг 7.6. Реализация `InstantClock` с фиктивным объектом для тестовых целей

```
public final class FakeInstantClock implements InstantClock {
    private final Instant currentInstant;

    public FakeInstantClock(@NonNull Instant currentInstant) {
        this.currentInstant = currentInstant;
    }

    public Instant getCurrentInstant() {
        return currentInstant;
    }
}
```

Конечно, кое-какие детали в этом листинге можно изменить. Например, задать в интерфейсе статический метод для получения фиктивного объекта и системных часов, сделав эти классы приватными в интерфейсе. Также можно предоставить `FakeInstantClock` с возможностью автоматического продвижения часов на конкретный промежуток при каждом вызове метода `getCurrentInstant()`. В данном случае важен сам способ использования часов, который предотвращает появление кода, непригодного для тестирования.

В приведенном коде почти нет комментариев, потому что он очень прост. Можно подумать, что от такого несложного кода особой пользы не будет, но он кардинально меняет тестируемость решения.

ПРИМЕЧАНИЕ

Возможно, вам интересно, почему мы вообще затеяли передачу фиктивных часов. В конце концов, проще имитировать интерфейс, содержащий единственный метод. Мы обнаружили, что имитации (mocks) чрезвычайно полезны для тестирования *взаимодействий*, где нас интересует, сколько раз и как вызываются методы интерфейса, но не для тестирования часов. Вместо этого мы просто предоставим данные, которые должны быть возвращены позже, а фиктивные объекты отлично подходят для этого. При желании можно использовать и имитацию, но, по нашему опыту, специально созданную фиктивную реализацию задействовать проще, и она отделяет код тестов от любой конкретной библиотеки имитаций.

Исключив один неявный источник информации с часами, проделаем нечто похожее для часовых поясов.

Предотвращение неявного использования системного часового пояса

Уровни поддержки абстракции часов в библиотеках даты и времени бывают разными, но на сегодняшний день любая из них, скорее всего, будет содержать

тип, представляющий часовой пояс. Однако многие методы все еще неявно используют системный часовой пояс, что приводит к тем же проблемам с тестированием. Не хотелось бы писать тесты, которые изменяют системный часовой пояс, запускают рабочий код, а затем возвращают часовой пояс к прежнему значению. Гораздо лучше просто сообщить коду, в каком часовом поясе он должен работать, даже если в окончательной версии всегда будет использоваться системный часовой пояс.

Конечно, *можно* написать две перегруженные версии (для соответствующего метода или конструктора типа): одна из них получает часовой пояс, а другая всегда использует системный. Но это чревато появлением кода со скрытой зависимостью от системного часового пояса. На расстоянии трех-четырех промежуточных уровней абстракции от этого конструктора или метода может быть неочевидно, что в выполняемых операциях задействован часовой пояс. Если всегда явно выражать свои намерения, то и сюрпризов не будет.

О возможных неожиданностях следует помнить и при вызове кода, за который вы не отвечаете. Это может быть как код самой библиотеки даты и времени, так и другая внешняя зависимость. И снова, возможно, придется поразмыслить, задействованы ли часовые пояса в конкретной операции или в коде по умолчанию используется системный часовой пояс.

Вернемся к примеру с политикой возврата: в системе может присутствовать метод для вычисления последней даты возврата для позиции заказа. В документе с требованиями уже упоминается часовой пояс, так что он очевидно должен присутствовать, но это не означает, что предоставить его должны вы. Для вычислений необходимы два исходных значения:

- момент времени, в который товар отправляется со склада;
- часовой пояс адреса доставки.

Часовой пояс уже задан контекстом операции, поэтому предоставлять его в методе, определяющем последнюю дату возврата, не обязательно.

Но когда дело доходит до написания кода, стоит снова подумать о простоте тестирования. Задать эти два значения в тесте несложно. Возможно, для формирования полного заказа из нескольких позиций потребуются больше усилий. Можно упростить модульные тесты, написав метод, получающий только эти два параметра, а затем вызвать его с моментом отправки и часовым поясом доставки. Этот метод не следует делать общедоступным, он должен быть видимым для тестирования.

В итоге в классе `OrderItem` остаются два метода: тривиальный открытый и более сложный внутренний, приведенный в следующем листинге. (Мы вернемся к фактической реализации позже.)

Листинг 7.7. Упрощение тестирования для сложного сценария

```

public LocalDate getFinalReturnsDate() {
    Instant shippingTime = getShippingDetails().getWarehouseExitTime();
    ZoneId deliveryTimeZone = getOrder().getDeliveryAddress().getTimeZone();
    return getFinalReturnsDate(shippingTime, deliveryTimeZone);
}

@VisibleForTesting
static LocalDate getFinalReturnsDate(Instant shippingTime,
    ZoneId destinationTimeZone) {
    // Реализация
}

```

Делегирование от открытого метода к внутреннему

Реализация

Если вы хотите разместить всю логику возврата в одном месте, сложный код можно вынести из класса `OrderItem`. В любом случае важны сигнатуры методов:

- в методе, который напрямую использует часовой пояс, последний уже известен, поэтому очевидно, какой из них выбрать;
- везде, где этот метод должен вызываться, часовой пояс необходимо *указать явно*, поэтому вероятность использования системного часового пояса довольно мала.

Когда речь заходит о выявлении кода, в котором может по умолчанию использоваться системный часовой пояс, стоит обращать особое внимание на перегруженные версии. Если вы вызываете метод, получающий часовой пояс в одной из таких версий, но не передаете его в аргументе, внимательно проверьте, какой часовой пояс будет использоваться по умолчанию. Даже если это тот пояс, который вам нужен, код будет понятнее, если выразить намерения явно. С системными значениями по умолчанию связан еще один важный аспект, который приведет нас к более масштабной теме текстовых представлений.

Неявные допущения относительно локального или культурного контекста

Интернационализация, локализация и глобализация (иногда для них используются обозначения `i18n`, `l10n`, `g11n`) — обширные темы, и здесь мы не сможем рассмотреть их в подробностях. Для работы с датой и временем необходимо понимать, что локальный контекст пользователя может влиять на два аспекта кода: дефолтную календарную систему и текстовый формат, используемый по умолчанию для представления даты и времени.

И хотя мы большей частью заменили системные часы для целей тестирования (а не потому, что они могут работать некорректно), с системным локальным контекстом дело обстоит примерно так же, как с системным часовым поясом: не стоит исходить из того, что локальный контекст, в котором вы хотите работать, соответствует локальному контексту системы.

Как правило, следует избегать операций, привязанных к конкретной культуре. Как уже говорилось, многим бизнес-инструментам достаточно григорианского календаря, даже если кто-то и пользуется другой календарной системой в личной жизни. Точно так же текстовые форматы неактуальны для большей части кода. По аналогии, выполняя арифметическую операцию с числами, мы не задумываемся, в какой системе счисления они записаны — в *десятичной* или *шестнадцатеричной*.

Вряд ли здесь можно дать конкретные рекомендации *по программированию*. Скорее нужно достаточно хорошо знать используемые библиотеки, чтобы понимать, когда вызываются методы, учитывающие культурный контекст. Возможно, стоит уделить больше внимания значениям по умолчанию, по крайней мере, для календарных систем. Если библиотека по умолчанию использует григорианский календарь и никакие другие календари поддерживать не нужно, скорее всего, код будет более читаемым, если придерживаться значения по умолчанию (вместо того, чтобы явно задавать григорианский календарь при каждом вызове). С другой стороны, если по умолчанию используется дефолтный календарь для системного локального контекста, я бы порекомендовал явно передавать его при вызове.

Область текстовых представлений *не ограничивается* культурными особенностями. Рассмотрим этот вопрос более подробно.

7.3.3. Текстовое представление даты и времени

Вы работаете с данными даты и времени с использованием хорошо абстрагированной библиотеки, которая выполняет всю черную работу за вас и старается защитить от ошибок. Все идет замечательно, пока вы работаете с данными в памяти. Но часто значения даты и времени нужно представить в текстовом виде — иногда для целей диагностики (например, журналирования и отладки) или для передачи данных между разными компьютерами (например, когда JavaScript в веб-браузере выдает запрос к серверу), а иногда при выводе данных пользователю. И в этом случае очень легко сбиться с верного пути и повернуть не туда. Как и в большей части этой главы, в этом разделе собраны вопросы, которые следует задать себе, а не одно идеальное решение, подходящее для любой ситуации. Начнем с почти философского вопроса: когда у вас есть текстовое представление, что это на самом деле *значит*?

Путаница текста и смысла

Когда вы видите строку, содержащую дату и время, не всегда понятно, что именно она представляет. Порой даже не ясно, что в строке не все очевидно; можно сделать неверные выводы просто из-за лишних предположений.

Один из самых ярких примеров такого рода — класс `java.util.Date`. Мы уже рекомендовали держаться от него подальше, но его текстовое представление — хорошее

учебное пособие, которое показывает, чего делать *не следует*. Возьмем следующую строку кода:

```
System.out.println(new Date());
```

Прямо сейчас на моем компьютере она выводит строку `Sun Dec 27 14:21:05 GMT 2020`. Даже если не обращать внимания на то, что имя `Date`, очевидно, неудачное и что при вызове неявно используются системные часы, посмотрим, какие выводы можно из этого сделать.

- Значение включает сокращение, похожее на аббревиатуру часового пояса: GMT. Можно предположить, что само значение учитывает часовой пояс.
- Значение включает день недели, сокращенное название месяца и год. *Можно* предположить, что значение учитывает календарную систему.
- Значение детализировано до секунд. Означает ли это, что конструктор `Date` был вызван точно в момент перехода между двумя секундами, или произошла потеря данных? Трудно сказать.

Наибольшую озабоченность вызывает первый пункт. На Stack Overflow можно найти массу вопросов о том, как преобразовать `java.util.Date` в другой часовой пояс, и понятно почему. В самом деле, класс `Date` представляет момент времени с миллисекундной точностью. Он не связан с часовым поясом *или* календарной системой. Метод `toString()` всегда использует григорианский календарь и системный часовой пояс по умолчанию, но они не являются частью самого значения. Названия месяца и дня не локализуются.

Использование системного часового пояса по умолчанию создает изрядную путаницу, но даже более разумная реализация `toString()` может сбить с толку. Допустим, вместо этого используется представление ISO-8601, и то же значение может выглядеть как `2020-12-27T14:21:05.123Z`, и вы будете получать тот же результат на любом компьютере в мире (конечно, если системные часы будут выдавать одинаковые значения). Пока вы не *знаете*, какое значение представляется, оно остается неясным. Вам не известна точность и не понятно, что означает Z, — то, что каждое значение будет выражаться в UTC или что тип представляемых данных включает другие смещения или часовые пояса. Мы не знаем, может ли представляемый тип данных использовать другие календарные системы.

Если ситуация кажется безнадежной, не отчаивайтесь. Я не пытаюсь отговорить вас от работы с текстовыми представлениями, а стараюсь подтолкнуть вас к тому, чтобы узнавать их ограничения. Что еще важнее, нужно понимать, какой тип данных представляется в текстовом виде — в идеале одна из концепций, представленных выше, или их сочетание (например, «дата и время со смещением UTC»). Какая точность доступна и должно ли это представление предотвращать потерю точности? Что необходимо знать, если вы хотите разобрать такое значение? Известен ли вам его *точный* формат?

Один из сценариев, заслуживающий особого внимания, — вывод информации в отладчике. В зависимости от используемого текстового формата отладчику ничто не мешает вывести значения двух переменных в одинаковых текстовых представлениях, а также показать, что эти значения не равны друг другу. Проблема не уникальна для даты и времени — аналогичные ловушки встречаются при работе с числами с плавающей точкой и даже с обычными строками. Просто помните: то, что вы видите в отладчике, может не отражать всей истины.

Итак, мы предупредили вас об опасностях излишнего доверия к текстовым представлениям. А теперь посмотрим, в каких областях преобразования могут создать проблемы на ровном месте.

Как избежать ненужных текстовых преобразований

Прозвучит банально, но стоит по возможности избегать преобразований в текстовую форму и из нее. Я видел много кода с конвертацией значений даты и времени в строки для целей, которые по сути и не являются текстоориентированными.

- Включение значения в запрос базы данных (непосредственно в SQL или в параметре).
- Преобразование между разными представлениями либо между типами в одной библиотеке (например, получение `LocalDate` из `LocalDateTime`), либо между разными библиотеками.
- Намеренная потеря информации — например, форматирование значения `LocalDateTime` без включения долей секунды и его последующий парсинг для усечения до второго уровня детализации.

В каждом из этих случаев прибегать к текстовым преобразованиям нежелательно по ряду причин.

- Они скрывают то, что вы пытаетесь сделать, поскольку используют обходные пути.
- Появляется риск случайной потери точности или других ошибок.
- Они почти всегда медленнее более прямолинейных решений.

Всякий раз, выполняя преобразование текста, стоит задать себе вопрос, действительно ли это текстоориентированная задача. Если это не так, подумайте, есть ли лучший подход. Это может потребовать немного больше усилий, но поможет сделать задачу более ясной, а работу — эффективной. Предположим, что мы провели исследование и решили, что нам действительно требуется преобразование текста. В этом случае есть еще несколько подводных камней, которых следует избегать.

Создание эффективных текстовых представлений

Я понимаю: может показаться, что мы прилагаем чересчур много усилий. В конце концов, легко просто вызвать `toString()` или его эквивалент для используемой платформы и покончить с этим. Но если потратить немного больше времени и тщательно обдумать желаемый результат, это может принести значительную пользу.

Возможно, стоит централизовать всю работу с текстом внутри заданного приложения; решить, как должны выглядеть результаты для каждой концепции и каждой аудитории; документировать результаты, написать код один раз и использовать его повсюду. Это гарантирует последовательность всего приложения и позволит избежать утомительных сеансов отладки. Конечно, вам все равно придется следить, чтобы в каждом случае использовался верный централизованный вариант, а это значит, что необходимо иметь четкое представление об аудитории.

Каждый раз, когда вы преобразуете дату и время в строку, представляйте, как эта строка будет читаться в будущем. Большинство ситуаций попадает в одну из трех категорий, показанных на рис. 7.9:

- текст выводится для пользователя;
- текст парсится кодом в другой системе;
- текст используется разработчиком в процессе диагностики.

У этих категорий разные мотивации и требования. Казалось бы, потребности пользователей и разработчиков должны быть схожими, но обычно сообщения для разработчиков (в журналах и исключениях) больше напоминают представления, предназначенные для машинного чтения.

	Аудитория?	Пример
Концептуальный уровень: дата и время со смещением UTC	Конечный пользователь	05/10/2021 4:30 p.m
	Разработчик	2021-05-10T16:30:00-05 (2021-05-10T21:30:00Z)
	Компьютер	2021-05-10T16:30:00-05

Рис. 7.9. Разный текст для разных аудиторий

В тексте, выводимом для пользователя, обычно должен учитываться локальный пользовательский контекст — по крайней мере, в отношении предпочтительного формата даты. Самый очевидный пример такого рода — принятые форматы числовых данных. В Соединенных Штатах используется формат «месяц/день/год», тогда как в большинстве стран мира используется формат «день/месяц/год». Кроме упорядочения, в различных локальных контекстах используют разные разделители компонентов даты и времени, а также форматы длинной даты (которые, к примеру, могут включать название месяца). Почти никогда не стоит пытаться прописывать точные форматы; многие библиотеки позволяют

указать общий формат (например, *короткий* или *длинный формат даты*), и этого будет достаточно для работы.

Разнообразие форматов означает, что почти никогда не стоит *парсить* текст, форматированный для пользователей. Возможно, вы захотите это сделать для анализа экранных данных, но это лишь одна из многих причин, по которым лучше такой анализ не проводить. Если вам *необходимо* парсить текст, видимый пользователю, постарайтесь передать локальный контекст той стороне, которая его создает. Если вы не знаете, что обозначает *6/7/2020* — *6 июля 2020 года* или *7 июня 2020 года*, — будет очень трудно выполнить необходимые действия без сложной и ненадежной эвристики.

Впрочем, с текстом для машинного чтения ситуация иная. Когда вы создаете текст, который будет читать другой компьютер, всегда старайтесь использовать стандартный формат. Для значений даты и времени это почти всегда означает совместимость с ISO-8601. Даже в рамках ISO-8601 доступно несколько форматов. Например, значение даты и времени 7 июня 2020 года в 15:54:23 можно представить как *20200607T155423.500*, *2020-06-07 15:54:23,5* или другими способами. При выборе представления используйте эти рекомендации:

- Если позволяет место, включите разделители компонентов даты и времени (косая черта и двоеточие соответственно) — это существенно упрощает чтение. Однако помните, что двоеточия запрещены в именах файлов Windows и могут плохо восприниматься в путях Unix, разделяемых двоеточиями.
- Необязательная буква T между датой и временем слегка затрудняет чтение значений, но помогает их сгруппировать. Это особенно важно, если контекст содержит несколько значений, разделенных пробелами.
- Хотя разделителем дробной части секунд может быть запятая или точка, а в ISO-8601 предпочтительной формой записи считается запятая, на практике точка встречается намного чаще.
- Отображение долей секунды до фиксированной длины может расходовать место, но упрощает чтение текста, если он состоит из столбцов с множеством значений. Если вы решите использовать переменную длину, придерживайтесь миллисекундной, микросекундной или наносекундной точности с 3, 6 или 9 знаками соответственно. Значения с 4 (например) знаками после точки или запятой выглядят довольно странно.

Возможно, вас удивит, что мы учитываем фактор удобочитаемости (для человека) — ведь эти значения предназначены для парсинга в коде. Но скорее всего, разработчики все равно будут просматривать текстовые файлы, запросы JSON или другие виды текстовой информации. В отдельных случаях приходится искать баланс между размером вывода, влияющим на каждое значение, и удобочитаемостью, актуальной только для одного значения из миллиона, но потери от плохо читаемых данных могут оказаться очень значительными.

Мы подходим к последней категории: разработчикам. Как правило, ориентированное на них текстовое представление должно быть культурно-нейтральным (как и машинно-читаемое). Но возможно, вы захотите дополнительно дать *необязательную* информацию. Я бы порекомендовал начать с простого представления ISO-8601 и добавлять информацию по мере необходимости. Например, если вы представляете дату и время со смещением UTC, можно включить местное время и момент UTC, чтобы упростить сравнение значений. В отдельных случаях наиболее подходящее для разработчика представление может содержать *меньше* информации, чем обычное. Например, при выводе записей журнала для приложения с коротким сроком жизни можно не включать дату, чтобы не перегружать вывод. А когда вы определитесь с текстовым представлением, остается написать его код.

Использование библиотек

Существует золотое правило обработки текста при операциях со значениями даты и времени (а на самом деле с большинством других текстовых представлений): не делайте это самостоятельно. Во всех нормальных библиотеках даты и времени реализована функция форматирования и парсинга данных, и она наверняка справится с задачей лучше вас, потому что это ее работа.

Из этого правила есть *одно* исключение: если представление достаточно неудобное и библиотека не может работать с ним напрямую без предварительной обработки текста. Представьте, что вам нужно парсить текст вида `Dec 28th 2020` в дату. Такое представление далеко не идеально для парсинга, но иногда нормальной альтернативы просто нет. В зависимости от используемой библиотеки могут возникнуть проблемы с порядковым числительным (`th` в `28th`). В этой ситуации лучше ограничиться минимумом преобразований для перевода текста в формат, подходящий для парсинга (например, `Dec 28 2020`), и воспользоваться библиотекой, чтобы парсить значение обычными средствами.

Внимательно прочитайте документацию по обработке текста для используемой библиотеки, особенно если вам придется работать с нестандартным форматом. Не стоит полагать, что форматные строки имеют абсолютно одинаковый смысл на всех платформах. Многие вопросы о дате и времени на сайте Stack Overflow, связанные с необъяснимыми ошибками парсинга даты и времени, объясняются тем, что разработчики недостаточно внимательно изучили шаблоны форматирования, особенно `m/M` (*минуты* и *месяцы*) и `h/H` (часы по *12-часовой* и *24-часовой* записи).

Как уже говорилось, обычно стоит централизовать хотя бы некоторые аспекты обработки текста даты и времени. Если вы обнаружите, что вам приходится задавать одну и ту же форматную строку в нескольких местах для одной цели, от такого дублирования определенно стоит избавиться. В зависимости от используемой библиотеки централизация может включать следующие компоненты:

- предоставление часто используемых, неизменяемых, потоково-безопасных объектов форматирования (таких, как `java.time.format.DateTimeFormatter` в `java.time` или `NodaTime.Text.LocalDatePattern` в `Noda Time`);
- предоставление методов для выполнения форматирования и парсинга;
- предоставление самих форматных строк (например, `"уууу-мм-дд'T'HH:mm:ss'Z'"` для формата момента времени согласно ISO-8601 с точностью до секунды).

Последний вариант прост, но не идеален с точки зрения безопасности типов. Вы рискуете использовать неправильную форматную строку, не подозревая о том, что вы пытаетесь отформатировать дату, как если бы она содержала дату и время. Тем не менее даже это гораздо лучше повторения одной форматной строки в нескольких местах.

В этом разделе чаще всего предполагается, что вы можете создать текстовый формат самостоятельно и таким образом получить полезное представление естественной моделируемой концепции. Но что делать, если такой возможности нет?

Преобразование данных в текстовом формате в концепцию

Иногда контролировать формат получаемых данных не удастся, из-за чего приходится работать с последствиями неудачных решений. Это может привести к ситуации, когда семантика значения не соответствует формату его представления.

Рассмотрим немного утрированный пример. Допустим, вы пишете приложение-будильник и хотите интегрироваться со сторонним сервисом для создания пользовательских сигналов, используемых во многих приложениях. Сигналы могут срабатывать ежедневно, а могут устанавливаться на конкретную дату. Они воспринимаются как разные значения — одно включает только время суток (которое в нашем приложении представляется классом `java.time.LocalTime`), а другое, к примеру, содержит дату и время (`java.time.LocalDateTime`). Логично ожидать, что они будут иметь разные представления в API, но это не обязательно. Можно получить разметку JSON следующего вида:

```
{
  "alarms": [
    {
      "dateTime": "2021-04-01T07:00:00",
      "type": "once",
      "label": "April Fool prank"
    },
    {
      "dateTime": "1970-01-01T06:00:00",
      "type": "daily",
      "label": "Wake up"
    }
  ]
}
```

Здесь время суток представляется полной датой и временем, а дата 1 января 1970 года становится данными, которые просто игнорируются, как показано на рис. 7.10.

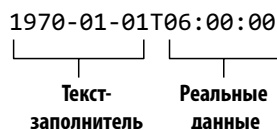


Рис. 7.10. Текстовые значения могут содержать текст-заполнитель вместе с реальными данными

Предположим, вы хотите использовать для представления времени суток в приложении `java.time.LocalDateTime`. Это можно сделать двумя способами:

- напрямую разобрать значение как `LocalTime` со специальным форматом, включающим литерал `1970-01-01T` перед частью времени;
- разобрать значение как `LocalDateTime`, а затем получить компонент времени суток; это просто сделать методом `toLocalTime()`.

Я бы порекомендовал второй вариант. Он разделяет операции *преобразования значения из текста в его естественное представление* и *получение действительно необходимого представления*. Кроме того, он позволяет напрямую моделировать JSON классом с использованием свойства `dateTime` типа `LocalDateTime`. Разметку JSON следует парсить, не обращая внимания на то, какая часть значения `dateTime` будет полезной, а затем преобразовать его в (потенциально) разные классы в зависимости от значения `type`. Два разных преобразования (JSON в объект и объект в объект) могут тестироваться независимо друг от друга. Каждый разработчик, который прочитает код модели JSON и увидит простой текст JSON, заметит прямое соответствие между ними.

Вот и все, что я хотел сказать о текстовых представлениях значений даты и времени. Это одна из тех тем, в которых всегда можно узнать что-то новое, но приведенные выше рекомендации должны помочь в эффективном решении всех возникающих проблем. Последний аспект, относящийся к реальному коду, вообще не влияет на поведение кода (с точки зрения компьютера), но может очень много значить для человека.

7.3.4. Объяснение кода в комментариях

В интернете бытует мнение, что оставлять комментарии к коду (касающиеся реализации) — значит признать свое поражение. Хотя я понимаю, откуда берутся такие взгляды, для меня они слишком радикальны. Конечно, я приношу ответственность, когда смысл кода становится ясным благодаря тщательному выбору

имен переменных, а также рефакторингу, который сохраняет краткость методов, и т. д. Но обычно все эти меры помогают понять, *что* делает код, а не *почему*.

В граничных случаях, не очевидных читателю кода, комментарии помогут объяснить, почему не был выбран очевидно более простой подход. Это особенно актуально для работы с датой и временем. Комментарии также пригодятся при написании тестов — они объяснят предназначение каждого из них. Вернемся к методу `getFinalReturnsDate`, реализуем его и включим пояснения по поводу реализации, как показано в следующем листинге.

Листинг 7.8. Подробные комментарии в коде

```
/**
 * Вычисляет последнюю возможную дату для возврата товара по простой
 * схеме нажатием кнопки. Вычисление базируется на дате отправки товара
 * со склада с учетом адреса доставки. Период возврата (в настоящее
 * время три месяца; см. {@link #RETURNS_PERIOD}) прибавляется к дате
 * отправки для получения предельной даты возврата. Если при добавлении
 * периода возврата день месяца переходит через конец месяца,
 * то результатом должно быть начало следующего месяца.
 *
 * @param shippingTime — Момент отправки товара со склада.
 * @param destinationTimeZone — Часовой пояс адреса доставки.
 * @return — Последняя дата возврата товара.
 */
@VisibleForTesting
static LocalDate getFinalReturnsDate(Instant shippingTime,
    ZoneId destinationTimeZone) {
    LocalDate shippingDateAtDestination =
        shippingTime.atZone(destinationTimeZone).toLocalDate();
    LocalDate candidateResult = shippingDateAtDestination.plus(RETURNS_PERIOD);
    // LocalDate.plus производит усечение при переполнении дня месяца.
    // Например, в java.time 31 марта+1 месяц дает 30 апреля, а не 1 мая.
    // В наших требованиях указано, что в таких случаях должен происходить
    // переход на следующий день. Для проверки подобных случаев проще всего
    // вычесть период возврата и посмотреть, вернется ли вы к исходной
    // дате отправки. Если нет, мы знаем, что произошло
    // переполнение и необходимо прибавить день.
    return
        candidateResult.minus(RETURNS_PERIOD).equals(shippingDateAtDestination)
            ? candidateResult
            : candidateResult.plusDays(1);
}
```

Несмотря на то что этот метод пакетно-приватный (и был бы полностью приватным, если бы мы не хотели использовать его напрямую для тестирования), Javadoc поможет подробно пояснить, что делает этот метод. Комментарий в реализации объясняет, почему мы вычитаем период возврата из результата: так проверяется наличие переполнения.

Очевидно, в отношении длины комментариев мнения расходятся. Приведенные выше комментарии можно немного сократить в зависимости от того, какой вариант кажется разработчикам понятнее. Возможно, в Javadoc-комментарий следовало бы включить ссылку на объявление открытого метода. Если полностью удалить Javadoc-комментарий, у нас все равно останется документ с требованиями, но он не объяснит, почему метод реализован именно так, а не иначе. Комментарий в реализации — ценная информация, которая нигде больше не отражена, и я бы не стал удалять его полностью.

Разработчики, которые не любят комментарии, считают тесты средством передачи информации, и я согласен с ними. В правильных тестах вряд ли будет нарушен граничный случай с переполнением дня месяца. Но при чтении кода вы наверняка не захотите пробовать что-то еще и смотреть, что сломается в программе, чтобы понять, почему код написан именно так. Раз уж речь зашла о тестах, рассмотрим некоторые из них для этого метода в следующем листинге.

Листинг 7.9. Комментарии в тестах для объяснения граничных случаев

```
public class OrderItemTest {
    private static Stream<Arguments> provideGetFinalReturnsDateArguments() {
        return Stream.of(
            // Простой случай: UTC делает дату очевидной, без переполнения.
            Arguments.of("2021-01-01T00:00:00Z", "Etc/UTC", "2021-04-01"),
            // Америка/Нью-Йорк - зимой UTC-5, потому время отправки
            // задается для 31 декабря 2020 года.
            Arguments.of("2021-01-01T00:00:00Z", "America/New_York", "2021-03-31"),
            // Переполнение дня месяца, пример приведен
            // в документе с требованиями.
            Arguments.of("2020-11-30T12:00:00Z", "Etc/UTC", "2021-03-01"),
            // Проверка часового пояса для адреса доставки: Америка/Нью-Йорк
            // переходит с UTC-5 на UTC-4 14 марта 2021 в 07:00:00Z.
            // В первом тесте отправка выполняется 13 марта,
            // а во втором - 15 марта 2021, несмотря на то что между моментами
            // отправки проходит ровно 24 часа.
            Arguments.of("2021-03-14T04:30:00Z", "America/New_York", "2021-06-13"),
            Arguments.of("2021-03-15T04:30:00Z", "America/New_York", "2021-06-15"));
    }
    @ParameterizedTest
    @MethodSource("provideGetFinalReturnsDateArguments")
    void getFinalReturnsDate(String shippingText, String zoneText,
        String expectedText) {
        Instant shippingInstant = Instant.parse(shippingText);
        ZoneId zoneId = ZoneId.of(zoneText);
        LocalDate expectedDate = LocalDate.parse(expectedText);
        LocalDate actualDate = OrderItem.getFinalReturnsDate(
            shippingInstant, zoneId);
        assertEquals(expectedDate, actualDate);
    }
}
```

Перед вами простой тестовый метод с пятью параметризованными тестами. Комментарий над каждым набором аргументов для тестового метода описывает, какой аспект метода проверяется данным тестом. Можно было написать пять разных тестовых методов с содержательными именами, но параметризация обычно обеспечивает бóльшую компактность и универсальность. В некоторых тестовых фреймворках для каждого списка аргументов можно предоставить описание, которое должно выводиться при сбое; не жалейте времени на изучение возможностей используемого тестового фреймворка. Точный механизм описания цели каждого теста не имеет значения, важно само наличие описания.

Также следует заметить, что эти тесты используют строки для параметров тестового метода, которые затем парсятся в методе. Такой подход может показаться немного странным после всех советов о применении наиболее подходящего типа данных в коде, но, по моему опыту, он значительно упрощает тесты. В последнем разделе этой главы рассматриваются некоторые граничные случаи, которые легко упустить из виду.

7.4. ГРАНИЧНЫЕ СЛУЧАИ

Все темы этого раздела как минимум *упоминались* ранее, но мы собрали их воедино как своего рода контрольный список, о котором стоит помнить. Все они должны учитываться в обычных приложениях — речь не идет о какой-то экзотике вроде корректировочных секунд. Начнем с ситуации, рассмотренной в примере с датой возврата: прибавления периода к дате.

7.4.1. Арифметические операции с календарями

Если вам нужен только григорианский календарь, как в большинстве приложений, вы, скорее всего, в курсе четырех потенциальных проблем, возникающих при выполнении арифметических операций с календарем.

- Високосные годы, из-за которых появляется 29 февраля, встречаются (приблизительно) через каждые четыре года.
- Переполнение дня месяца — например, при прибавлении месяца к 31 марта, потому что 31 апреля не существует.
- Ошибочные ожидания обратимости операций; в общем случае $(\text{date} + \text{period}) - \text{period}$ не всегда дает результат date .
- Ошибочные ожидания возможности упрощения; в общем случае $(\text{date} + \text{period1}) + \text{period2}$ не всегда дает результат, равный $\text{date} + (\text{period1} + \text{period2})$.

Просто знать об этих аномалиях часто бывает достаточно, чтобы упростить проектирование и тестирование. Рассмотренный ранее сценарий с голосованием достаточно типичен в том смысле, что в нем необходимо тщательно обдумать стратегии *добавления периода к начальной дате и проверки того, находится ли результат в прошлом, или вычитания периода из текущей даты и проверки того, что результат предшествует начальной дате*. Если для вас не принципиально, какой вариант выбрать, я обычно рекомендую выполнять календарные арифметические операции с фиксированным аспектом (то есть прибавлять период к начальной дате) — на мой взгляд, это проще для понимания и реализации.

Когда речь заходит о високосных годах, а также о тестировании, я настоятельно рекомендую избегать самостоятельной реализации логики «Является ли год x високосным?». В этом вопросе однозначно стоит довериться библиотеке даты и времени. Впрочем, есть и более общая рекомендация, для которой високосные годы являются лишь одним простым конкретным примером. Если вам придется выполнять особенно хлопотные операции с данными даты и времени, стоит выяснить, не реализован ли уже этот функционал в используемой библиотеке даты и времени.

Последнее, о чем стоит подумать касательно календарной арифметики — нужна ли она вообще? Зачастую *можно* работать с моментами и промежутками гражданских дат и периодов. Подумайте, действительно ли вас интересует затраченное время (что подразумевает использование промежутков) или же даты, которые важны людям (использование периодов.) Все остальные граничные случаи в этом разделе связаны с часовыми поясами — и это наверняка не удивит никого, кто работал с ними в серьезных инструментах.

7.4.2. Переходы часовых поясов в полночь

Как бы вы определили *полночь*? Есть два очевидных ответа: *12 часов ночи, то есть 00:00 по 24-часовой шкале, или время смены даты*. На первый взгляд это одно и то же, но не всегда.

В большинстве часовых поясов, соблюдающих летнее время, переключение происходит в 1 или 2 часа ночи по местному времени, но не везде. В некоторых случаях переход может привести к пропуску часа от 0 часов до 1 часа ночи или к возврату с 1 часа к 0 часов. В этом случае второе определение почти всегда происходит ровно один раз, но время 00:00 может наступить дважды, а может не наступить вообще.

Это значит, что если вы пытаетесь представить весь день в определенном часовом поясе, нужно определить, когда начинается конкретный день в этом поясе. Если предположить, что это происходит в 00:00, вы рискуете увидеть гору исключений в один день после перехода на летнее время. Я убедился в этом на своем горьком опыте. Проверьте, нет ли в используемой библиотеке даты и времени функции,

предоставляющей дату и время на начало этой даты в заданном часовом поясе. Если такой функции нет, то прежде чем использовать момент времени 00:00, необходимо проверить, является ли он действительным.

Это всего лишь один конкретный пример, в котором нужно учитывать фактор неоднозначного или пропущенного времени. *Обычно* проблема решается определением начала дня, но это решение не подходит для общего случая. Давайте разберемся почему.

7.4.3. Обработка неоднозначного или пропущенного времени

Как было показано выше, при обсуждении часовых поясов, из-за смещения UTC любая гражданская дата и время могут встречаться 0, 1 или 2 раза в заданном часовом поясе. (Это почти всегда связано с переходом на летнее время, но иногда стандартное смещение UTC часового пояса тоже может изменяться.)

Это может создать проблему как при указании конкретной даты и времени (*разбуди меня в 1:30 ночи 28 марта 2021 года в Лондоне*), так и при повторяющихся событиях (*проводить резервное копирование ежедневно в 1:30*). Между этими двумя примерами есть большие различия во взаимодействии с человеком: если пользователь вводит только дату и время, разумно запросить дополнительную информацию. Если речь идет о повторяющемся событии, возможно, нужно будет самостоятельно выбрать выполняемое действие. Скажем, в примере с резервным копированием можно запланировать его на 1:30 по местному времени (или позже). Таким образом, оно будет выполнено в 2 часа ночи (если часы переходят с 1 часа на 2 часа) или в момент более раннего наступления 1:30 ночи (если часы переводятся обратно). Это не единственный вариант, хотя, вероятно, самый простой для понимания. Здесь важно предвидеть такую возможность и принять соответствующее решение в требованиях и коде.

Подобные затруднения достаточно легко тестируются, по крайней мере, если применять абстракцию часов. Впрочем, лучше оставить в коде дополнительную информацию о часовом поясе, который используется для тестирования, — не стоит надеяться, что каждый разработчик, читающий код, точно знает время переходов во всех часовых поясах. Я также рекомендую использовать в тестах даты из прошлого, потому что прошлое обычно известно относительно хорошо (по крайней мере, теоретически), а будущее еще может измениться. Разберем эту тему подробнее.

7.4.4. Изменения данных часовых поясов

Ранее я упоминал о базах данных часовых зон Windows и IANA и о том, что они обновляются несколько раз в год, когда разные страны меняют правила часовых

поясов. Стоит пояснить: база данных не переделывается каждый раз, когда страна переходит с летнего времени на стандартное или наоборот. Подобные предсказуемые события предусмотрены правилами. Но это происходит при внесении изменений в сами правила, например:

- страна решает *прекратить* переходить на летнее время;
- страна решает *начать* переходить на летнее время;
- страна меняет время перехода на летнее время и обратно;
- страна изменяет стандартное смещение UTC.

В любом отдельном государстве такие изменения происходят относительно редко (по крайней мере, в общем случае). Но в мире много стран, поэтому база данных меняется несколько раз в год. Часто несколько изменений объединяются в один пакет; не стоит полагать, что каждый раз появляется отдельная версия базы данных. Прежде чем говорить о влиянии изменений часового пояса на код, стоит подумать, откуда ваше приложение получает данные часовых поясов.

Источники данных часовых поясов

Источник данных часовых поясов зависит от используемых платформ и библиотек. Например, с `java.time` платформа Java содержит встроенную версию базы данных часовых поясов, которая обновляется программой `TZUpdater`. Другие провайдеры правил часовых поясов могут регистрироваться при помощи класса `java.time.zone.ZoneRulesProvider`. Многие платформы загружают свои данные часовых поясов из операционной системы (обычно с возможностью ввести конкретную версию данных вручную).

Если вы пишете код клиентской стороны, который выполняется в браузере, данные часового пояса можно получить от браузера пользователя, если только вы не используете библиотеку, позволяющую загрузить конкретный набор правил. Это может привести к ситуации, в которой люди одновременно используют разные версии данных часовых поясов, что очевидно усложняет дело.

Изучите, а затем задокументируйте источники данных часовых поясов своего приложения, не забывая о том, что каждая используемая платформа может иметь отдельный источник. (Например, если одна часть кода работает в браузере пользователя, другая — в функции Node без сервера, а третья в сервисе .NET, все три источника документируются отдельно.) Как применяются обновления к этим данным и насколько вы это контролируете? Зная контекст, подумайте, как он влияет на данные приложения.

Хранение данных, чувствительных к изменениям часовых поясов

Выше мы говорили об источнике любых данных вашей системы. Это стоит повторить, чтобы сократить объем раздела; кроме того, подобное знание дает

некоторую уверенность. Любые метки времени, сохраняемые в системе, должны регистрироваться как моменты времени, *не зависящие* от часовых поясов. Момент закрепления транзакции в базе данных, размещения заказа или удаления данных пользователя не зависит от часового пояса. (Вероятно, он зависит от точности часов системы, в которой выполняется код, но это другой вопрос.) Во многих системах это справедливо для основного объема данных даты и времени.

Эти моменты могут быть преобразованы в местное время конкретного часового пояса в другой точке кода, но значение, которое хранится в виде момента, менять не нужно. Оно должно оставаться источником истинной информации, даже если где-то сохраняется другая производная информация.

Впрочем, для данных, которые вводит пользователь, часто справедливо обратное, особенно если они относятся к будущему. Здесь источником истинной информации являются местные дата и время, введенные пользователем, и его местонахождение или часовой пояс. Я часто вижу совет хранить *все* данные даты и времени в UTC, фактически преобразуя все моменты. Для данных, изначально базирующихся на моментах времени, это нормально, но в других ситуациях это может создать проблемы.

Простейший пример — планирование пользователем события в конкретном месте. На момент написания книги Франция соблюдает летнее время, так что в Париже используется смещение UTC+1 зимой и UTC+2 летом. Вполне возможно — и скорее всего так и произойдет, — что вскоре Франция полностью откажется от летнего времени и весь год будет использовать UTC+2. Рассмотрим пример со следующей последовательностью событий, в которой пользователь из Франции планирует встречу.

- 10 января 2021 года: пользователь планирует встречу на пятницу, 1 декабря 2023 года, на 9 часов утра в центре Парижа.
- 1 сентября 2021 года: правительство Франции объявляет, что с 27 марта 2022 года в 1 час ночи UTC Франция навсегда переходит на UTC+2.
- 27 ноября 2023 года: пользователь просматривает свое расписание на следующую неделю в приложении.

Что *видит* пользователь? Это должен решать владелец продукта, но я предполагаю, что почти во всех приложениях пользователь ожидает увидеть встречу в том виде, в котором она была запланирована: Париж, 1 декабря в 9 часов. В нашем мысленном эксперименте будем исходить именно из этого требования. (Впрочем, не забывайте об этом нюансе при работе над приложениями; универсального решения не существует.)

Допустим, когда пользователь планирует встречу, приложение преобразует дату и время в UTC, как рекомендуют многие разработчики. В 2021 году данные часового пояса связывают 2023-12-01T09:00 в Париже с 2023-12-01T08:00Z (где

z обозначает UTC). Когда пользователь проверяет свой календарь на 27 ноября 2023 года, приложение должно провести обратное преобразование, но в соответствии с актуальными данными часового пояса 2023-12-01T08:00Z в Париже отображается на 2023-12-01T10:00, так что пользователь уверен, что встреча запланирована на 10 часов. Последовательность событий изображена на рис. 7.11.

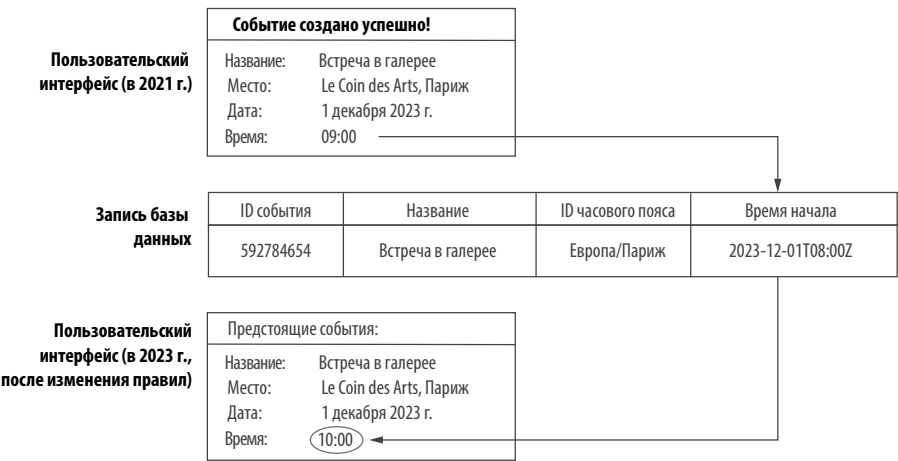


Рис. 7.11. Преобразование данных в UTC для хранения может вызвать проблемы

Если вы не хотите терять информацию от пользователя (местная дата и время в Париже), *сохраните* то, что он вам сообщил (местное время в Париже). Я понимаю, что это звучит банально, но это противоречит принятым правилам.

Хранить стоит далеко *не только* информацию, предоставленную пользователем. Часто бывает очень полезно сохранять значения UTC, чтобы сортировать записи в глобальном порядке. (Например, при сохранении значений UTC можно определить, наступает ли 5 часов утра в Калифорнии после 9 часов утра в Париже в один день.) Для этого необходимо отличать истинные данные (то, что сообщил пользователь) от производных (другая форма данных, вычисленная по истинным данным для простоты).

Когда истинные данные отличаются от производных, последние можно заново вычислить в любой момент — например, при изменении данных часового пояса. Конечно, при этом возникает еще один аспект: *версия* используемых данных часового пояса. Версия данных IANA формируется по простой схеме на основе года: 2020a, 2020b и т. д. Версия данных часовых поясов Windows не столь очевидна.

Если вам кажется, что все это добавляет сложности, я с вами соглашусь. Мы сохраняем местную дату и время, идентификатор часового пояса, дату и время

UTC и версию данных часовых поясов; также необходимо написать процесс обновления, который будет выполняться при каждом изменении информации часовых поясов. Вероятно, во многих приложениях — особенно хранящих данные из прошлого, среди которых изредка встречаются данные, относящиеся к будущему — об этом не стоит слишком беспокоиться ради соблюдения баланса точности и усилий по ее достижению. Тем не менее это должно быть сознательным решением с документированным обоснованием.

МИФ О ХРАНЕНИИ ВСЕХ ДАННЫХ В UTC

Идея о том, что все данные даты и времени следует хранить в UTC, — очень популярный миф даже среди опытных разработчиков. Многие из них не рассматривают возможность изменения правил часовых поясов. Наверняка вы встречали эту мысль в обсуждениях — в личных или в социальных сетях. Пожалуйста, привлекайте внимание к этой проблеме.

Выше я намеренно скрыл различие между истинными и производными данными. А вы его заметили? Мы начали говорить о *Париже*, но затем сохранили идентификатор часового пояса. Как насчет мест, в которых часовые пояса изменяются со временем? Кажется невероятным, но новые часовые пояса иногда *появляются* (часто из-за войн). Например, если страна делится надвое из-за гражданской войны, две образовавшиеся страны могут решить использовать разное местное время, так что два места, которые раньше *принадлежали* одному часовому поясу, теперь находятся в разных поясах.

Руководствуясь теми же рекомендациями, следует рассматривать идентификатор часового пояса как производные данные и следить, чтобы местоположение сохранялось как истинные данные. Тогда при изменении соответствия места и идентификатора часового пояса производные данные в базе данных тоже могут измениться. Узнать, когда меняется это соответствие, не так просто; дело не сводится к простому обнаружению изменений в данных часового пояса IANA, и детали могут зависеть от технологии, используемой для установления этого соответствия. Здесь мы снова возвращаемся к компромиссам: многие приложения могут решить, что обработка подобных изменений выходит за рамки необходимых задач. Соответствия часовых поясов меняются реже, чем правила для каждого часового пояса.

Говоря о достаточно редких проблемах и рискуя напугать вас, объединим этот раздел с предыдущим. Мы обсуждали повторное вычисление значений UTC на основе местных данных при изменении данных часового пояса. Чуть выше я упоминал о вариантах, связанных с соответствием местного времени и UTC, когда местное время выпадает или оказывается неоднозначным. Я говорил о том, что при вводе *проблемной* даты и времени у пользователя следует запросить дополнительную информацию. Хорошо жить в простом мире, в котором пользователь обращает на нас внимание и ему можно задавать вопросы. А если он

вводит дату и время *с учетом данных часового пояса* и они однозначны в момент ввода, но потом та же дата и время пропускаются или *становятся* неоднозначными из-за изменения данных часового пояса? Тогда приходится принимать решения относительно пользовательского ввода. *Можно* отправить электронное письмо с просьбой уточнить информацию, но это слишком большое усилие для редкой ситуации, которая находится внутри и без того редкого граничного случая. Впрочем, теперь вы знаете о проблеме и можете решить, какой способ вам больше по душе.

ИТОГИ

- Работать с данными даты и времени непросто. Тем не менее и с ними можно справиться, если подходить к делу ответственно и использовать подходящие инструменты.
- Данные даты и времени можно примерно разделить на концепции *машинного времени* (моменты и промежутки) и *гражданского времени* (календарные системы, даты, время суток, часовые пояса).
- Результаты календарных арифметических операций (например, добавления месяца к дате) могут быть неожиданными; это не всегда то же самое, что и простое целочисленное сложение.
- Во многих приложениях не нужно учитывать расширенные концепции, такие как корректировочные секунды и эффект относительности. Если вы с самого начала определите область применения требований, то сэкономите немало усилий.
- Требования к продукту в области даты и времени часто формулируются неоднозначно. Зафиксируйте поведение продукта на многочисленных примерах, включающих граничные случаи.
- На многих платформах разработки доступны разные библиотеки даты и времени. Уделите немного времени и выберите ту, которая удовлетворяет всем вашим требованиям и позволяет писать ясный, однозначный код.
- Последовательно применяйте концепции в своей кодовой базе, выполняя преобразования между представлениями только на границах систем.
- Применяйте абстракцию часов, чтобы код, использующий текущую дату и время, можно было нормально тестировать.
- Избегайте неявных зависимостей от системного часового пояса или системного культурного контекста. Там, где вы собираетесь их использовать, выразите их явно или внедрите в виде зависимостей.
- Текстовое представление значений даты и времени может быть разным в зависимости от контекста. Проанализируйте аудиторию, для которой

предназначена информация, и разработайте соответствующее текстовое представление.

- Иногда бывает трудно понять, почему код даты и времени написан именно так, а не иначе. Если вы уверены, что код отлично достигает поставленных целей, но при этом неясно, почему более простое решение не подходит (например, из-за граничных случаев), не бойтесь использовать комментарии для объяснения логики кода.
- Изменения часовых поясов (например, переход на летнее время) приводят к тому, что значения местной даты и времени либо пропускаются, либо оказываются неоднозначными. Подумайте (а также задокументируйте и протестируйте), как обрабатывать подобные сложные случаи.
- Правила часовых поясов меняются со временем. Подумайте, как приложение должно использовать обновленную информацию и как это повлияет на существующие данные, особенно на относящиеся к будущему.
- Преобразование местных значений в UTC в соответствии с данными часового пояса до сохранения иногда приемлемо, но может привести к потере данных из-за изменения правил. Будьте внимательны и не думайте, что это панацея!

8

Локальность данных и использование памяти

https://t.me/it_boooks

В этой главе:

- ✓ Локальность данных при обработке больших данных.
- ✓ Оптимизация стратегий соединения с использованием Apache Spark.
- ✓ Сокращение перетасовки данных.
- ✓ Использование памяти и дискового пространства при обработке больших данных.

В приложениях больших данных, использующих стриминговую и пакетную обработку, часто возникает необходимость использования данных из многих источников для извлечения аналитической информации и коммерческой ценности. Принцип локальности данных позволяет переместить вычисления к данным. Данные могут храниться в базе данных или файловой системе, и пока они помещаются на диске или в памяти компьютеров, все просто. Обработка может быть локальной и быстрой, но в приложениях больших данных огромные объемы данных не могут храниться на одном компьютере. Необходимо применять такие приемы, как *секционирование*, для распределения данных по нескольким компьютерам.

Размещение данных на нескольких физических хостах затрудняет извлечение полезной информации из данных, распределенных по нескольким точкам, доступным в сети. Соединение данных в таком сценарии — нетривиальная задача, требующая тщательного планирования.

Мы проанализируем процесс соединения данных в сценарии больших данных. Но прежде чем подробно рассматривать эту тему, для начала стоит разобраться с ключевой концепцией из области больших данных: *локальностью данных*.

8.1. ЧТО ТАКОЕ ЛОКАЛЬНОСТЬ ДАННЫХ?

Локальность данных играет важнейшую роль в обработке нетривиальных объемов данных. Чтобы понять, почему эта концепция решает многие проблемы, рассмотрим простую систему, в которой локальность данных не используется. Представьте, что у вас имеется конечная точка HTTP `/getAverageAge`, возвращающая средний возраст всех пользователей, которыми управляет сервис. На рис. 8.1 показано, как данные перемещаются к вычислениям.



Рис. 8.1. Перемещение данных к вычислениям для конечной точки HTTP `/getAverageAge`

Когда клиент выполняет этот вызов HTTP, сервис загружает все данные из хранилища. Это может быть база данных, файл или другая форма долгосрочного хранения. Когда все данные переданы сервису, он выполняет логику вычисления среднего: возрасты всех людей суммируются, подсчитывается их количество, после чего сумма делится на число пользователей. Это значение возвращается конечному пользователю. Важно заметить, что возвращается только одно число.

Этот сценарий можно назвать *перемещением данных к вычислениям*. Здесь следует сделать пару важных замечаний. Прежде всего, необходимо загрузить все данные, объем которых может составить десятки гигабайт. Пока эти данные помещаются в памяти компьютера, вычисляющего среднее, все нормально. Проблемы начинаются при работе с большими наборами, включающими терабайты или петабайты данных. В таких ситуациях перемещение всех данных на компьютер может оказаться слишком сложным или невозможным. Например, можно воспользоваться методом разбиения и обрабатывать данные пакетами. Второе важное наблюдение — необходимость отправки и получения больших объемов данных по сети. Операции ввода/вывода — самые медленные в обработке данных. В них задействовано чтение из файловой системы и блокирование, потому что передается большой объем данных. Существует достаточно большая

вероятность того, что некоторые пакеты будут потеряны и часть данных придется отправлять заново. Наконец, мы видим, что конечного пользователя не интересуют никакие данные, кроме результата вычислений (среднего значения).

Одно из преимуществ этого решения — простота его программной реализации (если исходить из того, что объем обрабатываемых данных помещается в памяти компьютера). Эти наблюдения и недостатки становятся главными причинами, по которым обработка в таких сценариях инвертируется и вычисления перемещаются к данным.

8.1.1. Перемещение вычислений к данным

Мы уже знаем, что отправка данных к вычислениям имеет ряд недостатков, а это может быть неприемлемо для больших объемов данных. Рассмотрим задачу из предыдущего раздела с применением локальности данных.

В рассматриваемом сценарии конечный пользователь видит ту же конечную точку HTTP `/getAverageAge`, ответственную за вычисление среднего. Используемая процедура сильно изменяется. Вычисление среднего — простая логика, но и она требует написания кода. Необходимо извлечь из записи каждого человека поле возраста, суммировать эти значения и разделить на их количество. Фреймворки обработки больших данных предоставляют API, который позволяет инженерам легко программировать такие преобразования и слияния.

Допустим, мы используем язык Java для программирования этой логики (или любой другой язык). Логика, ответственная за такие вычисления, создается в сервисе, но необходимо передать ее на компьютер, где хранятся фактические данные. Схема перемещения изображена на рис. 8.2.

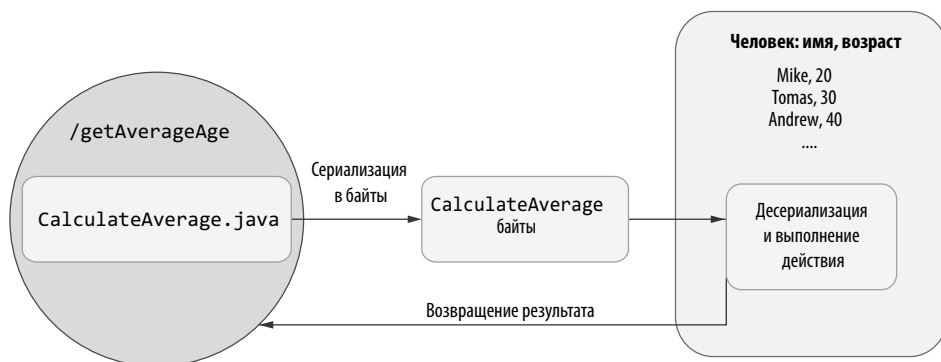


Рис. 8.2. Перемещение вычислений к данным и возвращение результата

Первое, что необходимо сделать, — сериализовать файл `CalculateAverage.java` в байты. Эта форма представления необходима для простой отправки данных по сети.

На узле данных (компьютере, на котором хранятся данные) должен выполняться процесс, отвечающий за получение сериализованной логики.

Затем байты преобразуются (десериализуются) в форму, которая может быть выполнена на узле данных. Многие фреймворки больших данных — такие, как Apache Spark или Hadoop, — представляют механизм сериализации и десериализации логики. После того как логика будет десериализована, она выполняется на самом узле данных. С точки зрения функции вычисления среднего логика оперирует данными, находящимися в локальной файловой системе. Нет необходимости отправлять данные любого человека сервису, предоставляющему конечную точку HTTP. Когда логика успешно вычислит среднее, сервису передается только итоговое число. Затем сервис возвращает данные конечному пользователю.

И снова в этом сценарии необходимо сделать некоторые важные замечания. Прежде всего, объем данных, которые нужно передать по сети, невелик; передается только сериализованная функция и полученное число. Поскольку сеть и ввод/вывод являются узким местом этой обработки, такое решение работает намного эффективнее: обработка с интенсивным вводом/выводом превратилась в обработку с интенсивными вычислениями. Если потребуются ускорить вычисление среднего, например, можно увеличить количество ядер на узле данных. При перемещении данных к вычислениям обработку ускорить сложнее, потому что не всегда можно повысить пропускную способность сети.

Решение, использующее локальность данных, получается более сложным, потому что в этом случае необходима логика сериализации обработки. При продвинутой обработке такая логика тоже может усложниться. Кроме того, потребуется специализированный процесс, работающий на узле данных. Этот процесс должен десериализовать данные и выполнить логику. К счастью, оба шага реализуются фреймворками больших данных (например, Apache Spark).

Некоторые читатели заметят, что тот же паттерн локальности данных применяется к базам данных. Если вы хотите вычислить среднее, то обращаетесь с запросом (например, на языке SQL), который передается базе данных. Затем база данных десериализует запрос и выполняет логику, использующую локальность данных. Эти схемы похожи, но фреймворки больших данных обеспечивают большую гибкость. Логика может выполняться на узлах данных, которые содержат любые виды данных: Avro, JSON, Parquet и любые другие форматы. Вы не зависите ни от какой исполнительной системы, привязанной к конкретной базе данных.

8.1.2. Масштабирование обработки с использованием локальности данных

Локальность данных играет критическую роль в обработке больших данных, потому что она позволяет легко масштабировать и распараллеливать обработку.

Представьте, что объем данных, хранящийся на узле данных, удваивается. В результате данные перестают помещаться в дисковом пространстве одного узла. Они не могут храниться на одной физической машине, поэтому распределяются на две (о том, как выполняется разбиение данных, рассказано в следующем разделе).

При использовании метода перемещения данных к вычислениям объем данных, которые необходимо передать по сети, удваивается. Обработка существенно замедляется, и ситуация только ухудшается, если узлов больше двух. На рис. 8.3 показано, как будут выглядеть данные после распределения по двум компьютерам.

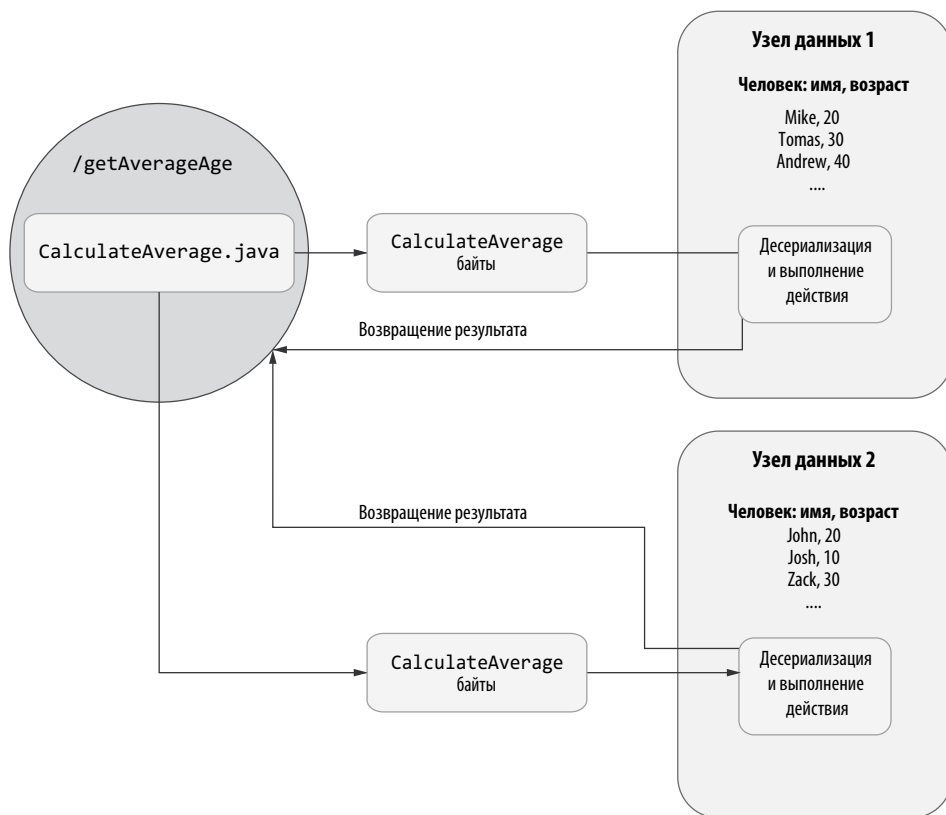


Рис. 8.3. Масштабирование обработки по двум компьютерам с использованием локальности данных

Масштабирование и распараллеливание обработки при использовании локальности данных достигаются достаточно легко. Вместо того чтобы отправлять сериализованную обработку на один узел данных, мы отправляем ее двум узлам. На каждом из них работает процесс, ответственный за десериализацию логики

и выполнение обработки. После того как обработка будет завершена, итоговые данные отправляются сервису, который собирает их воедино и возвращает конечным пользователям.

К этому моменту мы узнали преимущества локальности данных. Теперь необходимо понять, как разбить большие данные на N узлов данных. Это крайне важно, если вы хотите оперировать большими данными и извлекать из них ценность для бизнеса. Эта тема рассматривается в следующем разделе.

8.2. СЕКЦИОНИРОВАНИЕ И РАЗБИЕНИЕ ДАННЫХ

В предыдущем разделе было показано, что масштабирование обработки больших данных упрощается, если применить принцип локальности данных. В реальных приложениях больших данных объем информации, которую требуется хранить и обрабатывать, часто исчисляется сотнями терабайт или петабайт. Хранить такой объем данных на одном физическом узле нереально, необходимо разбить эти данные на N узлов. Метод разбиения данных, который рассматривается в этом разделе, называется *секционированием данных* (data partitioning), но это не единственный способ разбиения.

Для сетевых источников обработки данных (например, баз данных) можно выбрать идентификатор (например, идентификатор пользователя) и сохранить диапазон пользователей на выделенном узле. Например, если имеются 1000 идентификаторов пользователей и 5 узлов данных, на первом узле могут храниться идентификаторы от 0 до 200, на втором — от 201 до 400 и т. д.

К выбору схемы разбиения следует подойти внимательно, чтобы не создать *расфазировку данных* (data skew). Она может возникнуть, когда большинство данных создается по идентификатору или группе идентификаторов, принадлежащих одному узлу данных. Например, допустим, что идентификатор пользователя 10 отвечает за 80 % трафика и генерирует 80 % данных. Следовательно, это означает, что 80 % данных хранится на первом узле данных, так что секционирование не будет оптимальным. В худшем случае объем данных этого пользователя может оказаться слишком большим для хранения на одном узле. Важно, что при сетевой обработке секционирование оптимизируется для шаблонов чтения и записи данных.

8.2.1. Автономное секционирование больших данных

А теперь сосредоточимся на автономном секционировании обработки больших данных. В системах больших данных часто требуется хранить исторические данные («холодные» данные) в течение неопределенного времени. Очень важно хранить их максимально долго. Когда эти данные создаются, их коммерческая ценность может быть неочевидной.

Например, можно сохранять данные запросов всех пользователей вместе со всеми заголовками HTTP, но на момент их сохранения сценария использования заголовков HTTP может и не быть. Предположим, что в будущем вы решите создать инструмент, профилирующий пользователей по типу устройства (Android, iOS и т. д.). Такая информация распространяется в заголовках HTTP. Новая логика профилирования может выполняться на основании исторических данных, потому что они хранятся в необработанном виде. Важно заметить, что эти данные оставались ненужными долгое время.

А теперь допустим, что вы хотите сохранить большой объем информации на будущее. Таким образом, система хранения должна содержать большой объем данных в «холодном» хранилище. В приложениях больших данных это часто подразумевает сохранение данных в распределенной файловой системе Hadoop (HDFS). Также это означает, что секционирование данных должно быть достаточно общим. Мы не можем оптимизировать их для паттернов чтения, потому что пока не знаем, как будут выглядеть эти паттерны.

По этим причинам самая распространенная схема секционирования для автономной обработки больших данных основана на датах. Допустим, система сохраняет данные пользователя в каталоге файловой системы `/users`, а историю взаимодействий с ним — в каталоге файловой системы `/clicks`. Анализируется первый набор данных, в котором хранятся данные пользователя. Предположим, количество хранимых записей равно 10 миллиардам. Сбор данных начался в 2017 году и с тех пор не останавливался.

Выбранная схема секционирования основана на дате. Идентификатор секционирования начинается с года, поэтому, например, будут существовать секции для 2017, 2018, 2019 и 2020 годов. При небольших требованиях к данным разбиения по годам может быть достаточно. В таком сценарии данные пользователей хранятся в каталогах файловой системы `/users/2017`, `/users/2018` и т. д. (рис. 8.4); аналогичная схема используется и для данных о взаимодействиях: `/clicks/2017`, `/clicks/2018` и т. д.



Рис. 8.4. Четыре секции данных для схемы секционирования, основанной на дате

При таком секционировании данные пользователей будут состоять из четырех секций. Это означает, что данные можно разделить на четыре физических узла. В первом узле хранятся данные за 2017 год, во втором узле — данные за 2018 год и т. д. Ничто не мешает хранить все секции в одном физическом узле. Вариант с хранением в одном узле может быть вполне приемлемым — при условии, что места на диске достаточно. Когда оно будет исчерпано, мы создадим новый физический узел и переместим в него некоторые секции.

На практике такая схема секционирования недостаточно детализирована. Иметь одну большую секцию для всех данных за год нежелательно с точки зрения как чтения, так и записи. Когда вы читаете такие данные и вас интересуют только события за конкретную дату, вам придется сканировать данные за весь год! Это не только долго, но и неэффективно. Что касается записи, то, когда место на диске закончится, разбиение данных продолжать будет нельзя. Выполнить запись не удастся.

По этой причине в автономных системах больших данных обычно используется более детализированное секционирование данных по году, месяцу и дню. Например, если вы записываете данные за 2 января 2020 года, событие может быть сохранено в секции /users/2020/01/02. Такое секционирование обеспечивает большую гибкость и в отношении чтения. Если вы захотите проанализировать события за конкретный день, данные можно напрямую прочитать из секции. Если вы хотите провести высокоуровневый анализ (например, проанализировать данные за весь месяц), можно прочитать все секции за нужный месяц. Та же схема работает, если вы хотите проанализировать данные за целый год. Подведем итог: 10 миллиардов записей будут секционированы, как показано на рис. 8.5.

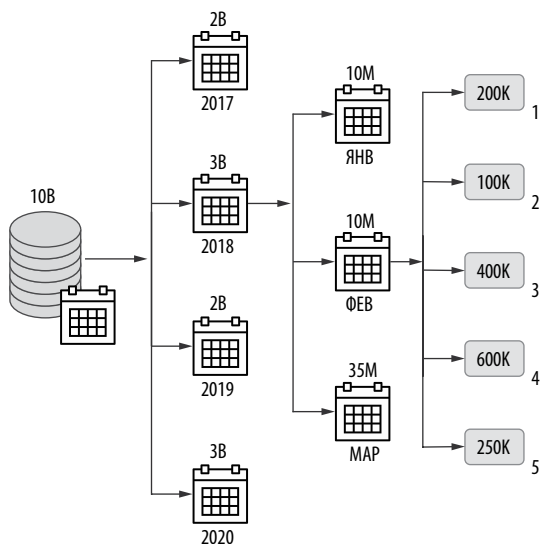


Рис. 8.5. Данные, основанные на дате, секционируются по году, далее по месяцу и затем по дню

Из диаграммы видно, что исходные 10 миллиардов записей секционируются по году, месяцу и, наконец, по дню месяца. В итоге секция каждого дня содержит 100 000 записей. Такой объем данных может легко уместиться на диске одного компьютера. Также это означает, что для одного года создаются 365 или 366 секций. Верхний порог узлов данных равен количеству дней, умноженному на количество лет, за которые хранятся данные. Если данные за один день не умещаются на диске одного компьютера, их можно секционировать дальше по часам, минутам, секундам и т. д.

8.2.2. Секционирование и сегментирование

Если предположить, что данные секционируются по дате, их можно разделить по нескольким узлам. В таком сценарии на физическом узле хранится подмножество всех ключей секций.

Допустим, данные пользователя разбиваются на N секций (логических сегментов), а детализация секционирования составляет 1 месяц. В таком случае данные за 2020 год состоят из 12 секций, которые могут быть разбиты горизонтально на N физических узлов (физических сегментов (shards)). Важно заметить, что N меньше либо равно 12. Иначе говоря, максимальное количество физических сегментов равно 12. Этот архитектурный паттерн называется *сегментированием* (sharding).

Теперь допустим, что физических узлов три. Тогда можно сказать, что данные пользователя за 2020 год разбиваются на 12 секций. Затем данные распределяются по трем сегментам (узлам). Каждый из узлов хранит четыре секции за 2020 год ($12 \text{ секций} \div 3 \text{ узла} = 4 \text{ секции на узел}$), как показано на рис. 8.6.



Рис. 8.6. Сегментирование для 3 физических узлов и 12 секций

На диаграмме физический сегмент эквивалентен физическому узлу. Ключи секций (логические сегменты) распределяются равномерно по физическим

сегментам. При добавлении нового узла в кластер каждый сегмент должен переназначить один из своих логических сегментов в новый физический узел.

Существуют различные алгоритмы назначения сегментов. Они также должны обрабатывать перераспределение сегментов в случае добавления или удаления узлов (при сбоях или понижающем масштабировании). Этот подход используется большинством технологий и хранилищ больших данных — таких, как HDFS, Cassandra, Kafka, Elastic и других. Подробности выполнения сегментирования зависят от реализации.

8.2.3. Алгоритмы секционирования

Описанная выше методика называется *диапазонным секционированием* (range partitioning), при котором данные делятся на диапазоны в зависимости от даты их генерирования. В зависимости от паттернов чтения можно выбрать другой способ секционирования данных.

Допустим, вы хотите получить все события, сохраненные для конкретного идентификатора пользователя. При использовании диапазонного секционирования сделать это достаточно сложно. Чтобы загрузить всех пользователей для заданного идентификатора, нужно сканировать все секции, которые могут существовать на разных физических узлах, а затем отфильтровать данные, представляющие интерес. Мы не сможем воспользоваться локальностью данных. Необходимость сканирования всех секций объясняется тем, что мы не знаем заранее, когда было выполнено действие с конкретным `user_id`. Оно может находиться в секции любой даты.

Допустим, данные требуется секционировать по значению `user_id`. Необходимо равномерно распределить N ключей по M физическим узлам. Проверенный метод для решения этой задачи основан на алгоритме хеш-секционирования. Сначала `user_id` хешируется некоторым алгоритмом хеширования (например, MurmurHash), который возвращает число. Затем с числом определяется операция вычисления остатка от целочисленного деления на M (где M — количество узлов). В идеале каждый узел должен содержать $N \div M$ секций. Можно пропустить хеширование и выполнить операцию прямо с идентификатором пользователя при условии, что он является числом. Но чтобы этот алгоритм работал для любого типа ключей секционирования (например, для `String`), хеширование применяется для преобразования нечислового значения в числовое. Это преобразование представлено на рис. 8.7.

В нашем примере используются два узла ($M = 2$). Допустим, им назначены идентификаторы 0 и 1. Когда приходит первое событие для `user_id` 1, мы применяем к этому идентификатору хеш-функцию, а затем выполняем с результатом операцию вычисления остатка от целочисленного деления на 2.

Для `user_id 1` результат будет равен 1. В результате это событие отправляется узлу с идентификатором 1 и сохраняется в нем. При поступлении второго события (на этот раз для `user_id 2`) алгоритм секционирования распределяет его в узел 0 и сохраняет там. Далее приходит другое событие для `user_id 1`. Алгоритм секционирования определяет, что он должен попасть в узел с идентификатором 1.

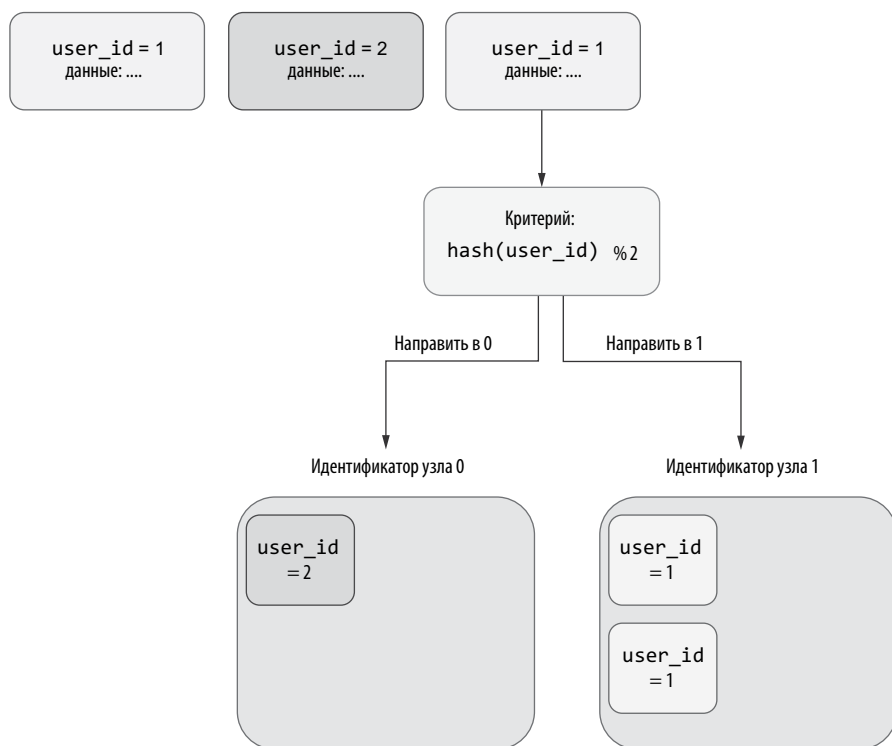


Рис. 8.7. Хеш-секционирование, используемое для разбиения данных по значению `user_id`

Такое поведение гарантирует, что все события для `user_id 1` будут сохраняться в одном узле. Благодаря этому можно легко осуществлять операции по `user_id`, использующие локальность данных. Данные этого пользователя уже находятся в одном узле. С этим алгоритмом все N значений `user_id` будут равномерно распределены по двум узлам.

Представленное решение — отличный пример для понимания секционирования, но у него есть пара недостатков. Главная проблема связана с тем, когда мы решаем добавить новый узел в кластер. Аналогичная трудность возникает при удалении одного из узлов (намеренно или вследствие сбоя).

Рассмотрим ситуацию с добавлением нового узла (рис. 8.8). Внезапно алгоритм секционирования изменяется, потому что приходится вычислять остаток от деления на 3 (количество узлов).

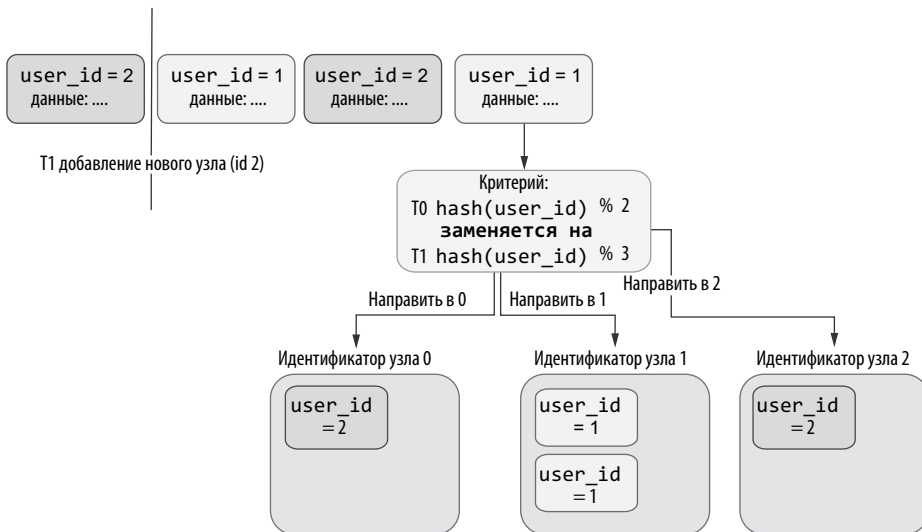


Рис. 8.8. Добавление нового узла

Допустим, первые три события отправляются, когда узла данных два (так что назначение секций происходит так же, как в приведенном выше примере). Обозначим это состояние T_0 (время 0). Затем во время T_1 в кластер добавляется новый узел. Схема распределения секций внезапно меняется, потому что остаток должен вычисляться в результате деления на 3. При появлении `user_id 2` новый алгоритм секционирования вычисляет, что для этого идентификатора секция равна 2. В результате новое событие отправляется на новый узел. Очевидно, при этом теряется локальность данных: события для одного идентификатора пользователя (2) теперь хранятся на двух физических узлах. Схема секционирования нарушается с точки зрения желаемого результата.

Как с этим справиться? Можно перенести все события для `user_id = 2` на новый узел, которому принадлежит идентификатор. Однако с наивной схемой вычисления секций, основанной на количестве узлов, число таких операций может быть существенным. Оно подразумевает много перемещений данных, что стоит достаточно дорого. Продемонстрируем вычисления для 10 идентификаторов. При делении на 2 идентификаторы 1, 3, 5, 7 и 9 направляются на узел с идентификатором 1, а значения 2, 4, 6, 8 и 10 — на узел с идентификатором 0.

Как изменится эта схема при добавлении нового узла (деление на 3)? Идентификаторы 3, 6 и 9 будут направляться на узел 0. Идентификаторы 1, 4, 7 и 10

приходятся на узел 1, а остальные идентификаторы (2, 5 и 8) — на узел 2. Только идентификаторы 3, 6 и 9 сохраняют прежний физический узел, а остальные (70 % данных) нуждаются в перераспределении.

В реальном кластере этот эффект умножается на количество узлов. Чем больше узлов, тем больше перемещений данных может потребоваться. Если на узлах хранятся большие объемы данных, такой процесс перераспределения может оказаться невыполнимым за разумное время. Что еще хуже, это может повлиять на работу сетевого приложения, которое сохраняет данные в таком хранилище данных.

Для снижения этого эффекта можно рассмотреть алгоритм консистентного хеширования (<http://mng.bz/Yg9B>). Он решает проблему введением виртуальных слотов, которые соответствуют M узлам. При добавлении нового узла необходимость в перераспределении возникнет лишь для малой части виртуальных слотов. Во многих реальных системах используются различные модификации этого алгоритма.

Теперь, когда вы узнали о принципах локальности данных и понимаете, как происходит секционирование данных, попробуем решить проблему соединения наборов данных из нескольких секций, размещенных на разных физических машинах. Решение этой проблемы приведено в следующем разделе.

8.3. СОЕДИНЕНИЕ НАБОРОВ БОЛЬШИХ ДАННЫХ ИЗ НЕСКОЛЬКИХ СЕКЦИЙ

Проанализируем три разных бизнес-сценария использования, требующих трех разных стратегий соединения. Каждая из этих стратегий задействует локальность данных на том или ином уровне. Мы проанализируем эти сценарии на концептуальном уровне, пока не погружаясь в подробности реализации — они будут рассмотрены в следующем разделе.

Начнем с общей структуры хранимых данных. Как вы помните, мы разделили их на секции в зависимости от даты и используем два источника данных. Первый источник данных хранит данные пользователя. Каждая секция (например, `users/2020/04/01`) содержит N файлов с N пользовательскими записями. Данные могут храниться в произвольном формате, текстовом или двоичном.

Например, если выбран двоичный формат AVRO, пакет записей будет сериализован в этот формат и сохранен в файле на HDFS. Одна секция может содержать N файлов. Каждый из этих файлов будет содержать часть данных для заданной секции. Как правило, один файл занимает не больше максимального размера блока в файловой системе. Для HDFS это значение составляет 128 Мбайт.

Например, если в `users/2020/04/01` хранятся 200 Мбайт данных, будут созданы два файла: `users_part1.avro` и `users_part2.avro` (рис. 8.9). Каждый файл содержит

данные пользователя. Предположим, что для каждого пользователя хранятся несколько значений (например, возраст и имя). Но самое важное, что у пользователей имеется идентификатор `user_id`, который однозначно определяет пользователя. Он будет использоваться при выполнении операции соединения с другими наборами данных.

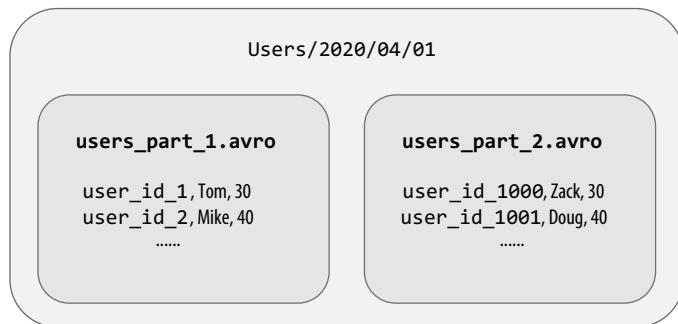


Рис. 8.9. Данные пользователей в секции

То же самое происходит с разделом `clicks`. Он содержит N файлов для секции, отвечающей за конкретную дату. Например, `clicks/2020/01/20` может содержать файлы с записями для заданной секции. В сценарии с соединением важно заметить, что данные `clicks` также содержат поле `user_id`, что позволяет связать данные взаимодействий с данными конкретного пользователя из набора пользовательских данных. Эти отношения будут использоваться при выполнении соединения с двумя наборами больших данных.

8.3.1. Соединение данных на одной физической машине

Первый сценарий, который мы рассмотрим, — соединение данных о взаимодействиях и пользовательских данных для одной даты. Иначе говоря, необходимо получить все данные о взаимодействиях во время посещения пользователем за конкретный день.

Это довольно распространенный сценарий использования: соединение данных, сгенерированных разными системами для одного идентификатора пользователя. Можно представить, что данные пользователя содержат информацию о платежах, транзакциях и других операциях, которые выполняет пользователь. Они могут собираться сервисами, отвечающими за платежи и завершение действий пользователей. С другой стороны, данные о взаимодействиях содержат менее чувствительную информацию; каждый клик на веб-сайте собирается и сохраняется сервисом взаимодействий. Эта информация может использоваться для отслеживания закономерностей использования и активности пользователя. Наша цель — связать

данные о взаимодействиях с действиями пользователей. Ведет ли взаимодействие к конкретной транзакции? А может, пользователь часто кликал по ссылкам, но в итоге отказывался от покупки? Соединяя данные, можно извлечь из них больше полезной информации, которая создает ценность для бизнеса.

Допустим, вы хотите соединить данные пользователей и кликов за весь 2020 год. Обработку можно распараллеливать до 366 (дней), так как это доступное количество секций за год. Процесс должен пройти по всем записям секции заданного пользователя, найти соответствующие данные в секции `clicks`, а затем соединить данные с использованием идентификатора `user_id`.

Будем считать, что на одной физической машине хранятся данные одной секции даты как для данных пользователей, так и для данных о кликах. Например, при соединении данных за 2020/01/01 данные кликов и пользователей будут локальными для процесса. Таким образом, можно воспользоваться локальностью данных. Нет необходимости загружать данные из удаленного источника. Каждый фрагмент данных, необходимый для выполнения соединения, присутствует на узле данных.

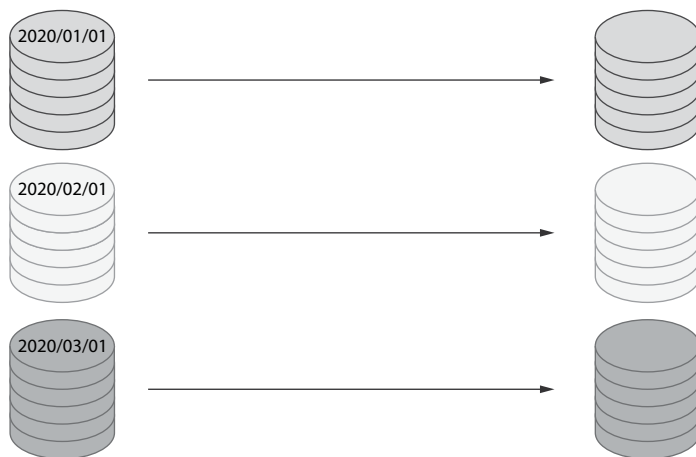


Рис. 8.10. Узкое преобразование без перемещения данных при соединении двух наборов данных

Допустим, все секции для конкретного месяца находятся на одном физическом узле данных. При выполнении соединения за январь 2020 года все данные о кликах за первый день месяца доступны локально, так же как и данные пользователя. Соединение данных в экосистеме больших данных можно описать как *преобразование* двух наборов данных в один итоговый набор. Преобразование называется *узким*, если его выполнение не требует перемещения данных. Другими словами, данные можно полностью преобразовать (выполнить соединение) с использованием локальности данных. Далее мы рассмотрим бизнес-сценарий, требующий операции соединения с перемещением данных.

8.3.2. Соединение, требующее перемещения данных

Следующий рассматриваемый сценарий требует соединения данных между секциями (рис. 8.11). Допустим, вы хотите найти всех уникальных пользователей с 2020 года. Это означает, что данные из секций всех месяцев требуется соединить по `user_id`. После соединения данных необходимо оставить только одно значение на каждый идентификатор пользователя, удалив дубликаты.

Важно заметить, что в этом сценарии нужно обработать все события пользователей для всех секций пользователей. Например, рассмотрим сценарий соединения данных для `user_id` 1. Сначала необходимо отфильтровать пользователей с этим идентификатором на всех узлах данных. Все секции разделов для 2020/01 могут выполнить логику фильтрации, используя локальность данных. Эту операцию следует выполнить с каждой секцией даты. Когда данные будут отфильтрованы, их необходимо отправить на узел данных, обрабатывающий `user_id` 1. При этом все еще предполагается, что данные каждого месяца размещаются на выделенном узле данных. Это означает, что данные всех 12 месяцев 2020 года должны быть отправлены на узел, выполняющий операцию соединения для идентификатора пользователя. На практике каждый узел данных будет обрабатывать диапазон идентификаторов.

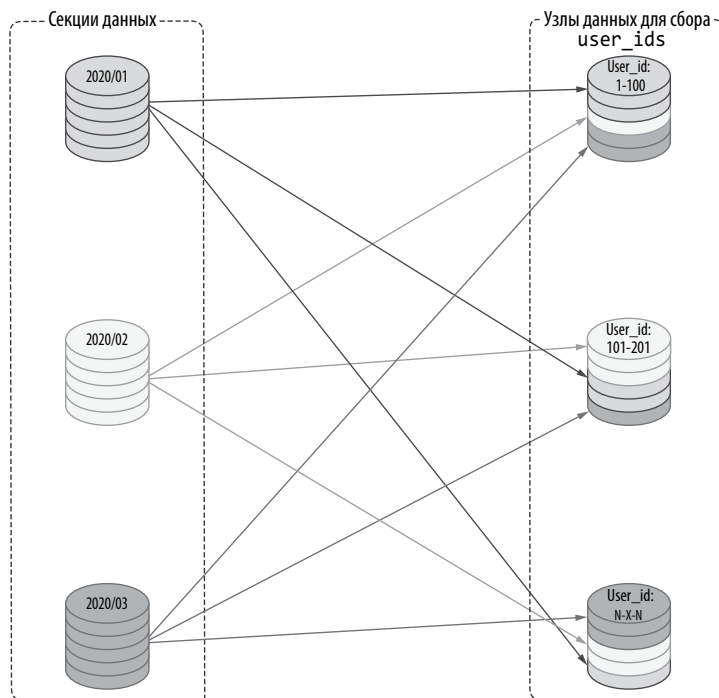


Рис. 8.11. Широкое преобразование с перемещением данных между секциями

Эта логика соединения требует значительного перемещения данных на втором этапе. Хотя мы используем локальность данных на этапе фильтрации, второй этап требует пересылки данных по сети. Преобразование, требующее перемещения данных, называется *широким*. Процесс обмена данными между узлами данных называется *перетасовкой* (shuffling). Чем больше перетасовка данных при обработке больших данных, тем медленнее она происходит. Как уже говорилось, операции, требующие значительной передачи данных по сети, далеко не оптимальны.

Некоторые оптимизации позволяют сократить перетасовку данных при выполнении соединений. Тем не менее они сильно зависят от сценариев использования и характеристик данных. При соединении данных часто оказывается, что один из наборов данных меньше, а другой намного больше. В таких случаях можно реализовать гибридное решение, использующее локальность данных в максимально возможной степени. С другой стороны, данные также требуют перетасовки, но ее можно сократить до минимума.

8.3.3. Оптимизация соединения за счет широковещательной рассылки

Теперь рассмотрим сценарий, позволяющий реализовать полезную оптимизацию соединения. Согласно бизнес-сценарию, нужно получить данные о кликах только за один месяц, поскольку требуется найти корреляцию между одним из клиентов пользователя и изменениями данных пользователя, которые произошли недавно (то есть в текущий день обработки).

Такой процесс выполняется каждый день для сопоставления кликов за текущий день по всем пользователям. Процесс соединения будет выполняться для пользователей за текущий год. Чтобы продемонстрировать эту оптимизацию, внесем одно дополнительное изменение в исходный пример. Как выяснилось, данные о кликах и пользовательские данные занимают слишком много места на диске, и оба источника данных необходимо переместить на отдельную физическую машину. Это значит, что локальность данных пользователей и данных о кликах исчезает. Все операции, при которых необходимо соединение этих источников данных, требуют перемещения данных.

Есть одно важное наблюдение, о котором следует упомянуть особо. Данные о кликах, используемые для соединения, содержат всего один день данных; это означает, что размер набора данных относительно невелик. Однако данные пользователей, которые должны участвовать в соединении, довольно объемны. Для реализации описанного сценария необходим целый месяц пользовательских данных. В результате мы получаем сценарий, в котором один набор данных в соединении на порядки меньше другого.

При выполнении соединений главной целью становится сокращение перетасовки данных за счет максимального использования локальности данных. Обеих целей можно достичь, отправляя меньший набор данных (клики) на узел данных, содержащий больший набор данных (пользователи), так что в этом сценарии мы загружаем данные кликов и распространяем их на все узлы данных, содержащие данные пользователей за 2020 год (рис. 8.12). Не забывайте, что для каждого месяца существует специализированная физическая машина. Это означает, что данные о кликах нужно распределить по 12 машинам.

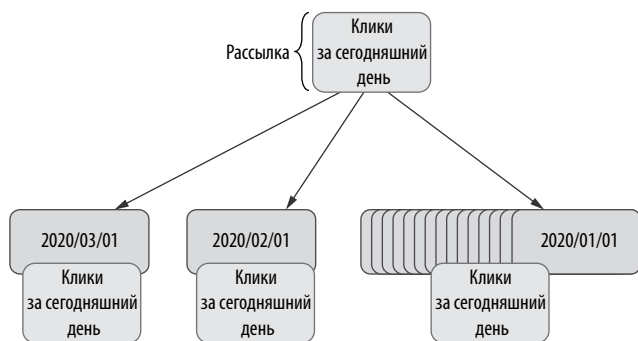


Рис. 8.12. Рассылка с соединениями

Набор, содержащий данные за сегодняшний день, рассылается по всем узлам данных, содержащим данные пользователя. Назовем его *рассылаемым набором данных*. Нагрузка на сеть (перетасовка данных) требуется только для отправки меньших наборов данных на узлы, на которых находится больший набор. Так как процесс соединения выполняется локально на машинах с данными пользователей, процесс проверит секции всех дней за январь 2020 года и соединит данные с кликами за сегодняшний день. Операция повторяется для всех месяцев 2020 года.

Этот прием позволяет задействовать локальность набора данных пользователей. По сети достаточно передавать только небольшую часть данных о кликах. Впрочем, здесь нужно сделать одно важное предупреждение. Такой прием оптимизации правильно работает лишь в том случае, если меньший набор данных полностью помещается в памяти узла данных. Когда данные окажутся в памяти, к ним можно обращаться на порядки быстрее, чем к данным на диске или к данным, которые необходимо передавать по сети. В следующем разделе описаны компромиссы между обработкой больших данных, использующей память (например, Apache Spark), и более старыми дисковыми решениями (например, Hadoop).

Следует заметить, что, хотя мы используем Spark, описанные приемы типичны для большинства фреймворков обработки больших данных. Независимо от API, предоставляемого такими фреймворками, они обычно используют парадигму MapReduce. Следовательно, оптимизация, рассмотренная в следующем

разделе, применима ко всем этим фреймворкам. Посмотрим, как это повлияет на быстроедействие операции соединения.

8.4. ОБРАБОТКА ДАННЫХ: ПАМЯТЬ И ДИСК

К этому моменту вы узнали, как использовать локальность данных в соединениях. Это позволяет сократить время обработки за счет снижения объема данных, передаваемых по сети. Тем не менее, даже если мы обрабатываем локальные данные, нужно загрузить их во фреймворк больших данных, выполняющий соединение.

Немного расширим пример из предыдущего раздела. Как вы помните, меньший набор данных (клики) соединялся с большим (пользователи). Набор данных о кликах передавался по сети на узел, выполняющий обработку и содержащий данные пользователя. Переданный набор сохранялся в памяти узла данных.

8.4.1. Обработка с хранением данных на диске

Теперь посмотрим, что произойдет с данными пользователя. Предполагается, что они не помещаются в памяти компьютера, поэтому их необходимо читать с диска по мере обработки. Проблема можно решить двумя способами.

Первое решение основано на отложенном чтении фрагментов файлов. Допустим, данные пользователя занимают 100 Гбайт и разбиты на 1000 частей. Каждая часть занимает 100 Мбайт, для нее на диске создается специальный файл. Когда процесс соединения завершает обработку первого фрагмента файла, он записывает результат в промежуточный файл. После этого он продолжает обработку: загружает следующую часть данных, выполняет новое соединение и сохраняет их снова. Процесс повторяется, пока не будут обработаны все данные (рис. 8.13).

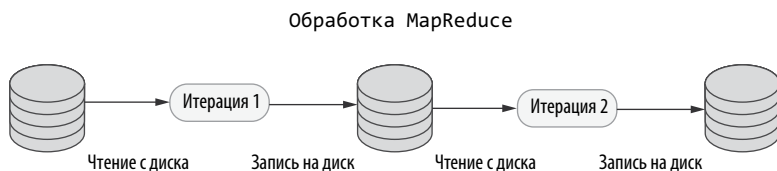


Рис. 8.13. Обработка больших данных с их хранением на диске

Собственно, стандартная обработка больших данных MapReduce на базе Hadoop работает именно так. Обработка Hadoop построена на обращениях к диску и тесно связана с файловой системой. Главной точкой интеграции между этапами обработки больших данных является файл. Результаты каждого этапа сохраняются в HDFS (файловой системе). Следующая задача Hadoop берет эти данные и обрабатывает их.

У такого подхода есть преимущества. Он позволяет разработчикам писать свои части процесса обработки независимо. На каждом этапе обработки получаются неизменяемые результаты, которые нельзя корректировать. Входными данными на каждом этапе обработки служит путь в файловой системе, а выходными — файлы в другом месте. К сожалению, у обработки больших данных на диске есть один колоссальный недостаток — она невероятно медленная.

8.4.2. Для чего нужна парадигма MapReduce?

В основу MapReduce заложена идея локальности данных. Чтобы понять, почему нужны этапы отображения (map) и свертки (reduce) и как в MapReduce используется локальность данных, рассмотрим решение известной задачи подсчета слов. Для этого мы воспользуемся парадигмой MapReduce и попробуем понять, почему это лучшее решение для больших наборов данных.

Допустим, N текстовых файлов распределены по M узлам данных в кластере. Каждый текстовый файл представляет собой набор больших данных и занимает N Гбайт. Требуется подсчитать вхождения каждого слова в каждом текстовом файле. Важно, что все наборы данных (M узлов $\times N$ Гбайт на файл) не помещаются в памяти на одном компьютере. Следовательно, обработку необходимо распределить.

Сосредоточимся на первом этапе обработки — локальном в отношении локальности данных. Все операции выполняются в контексте узла, из которого поступили данные. Первый этап процесса представлен на рис. 8.14.

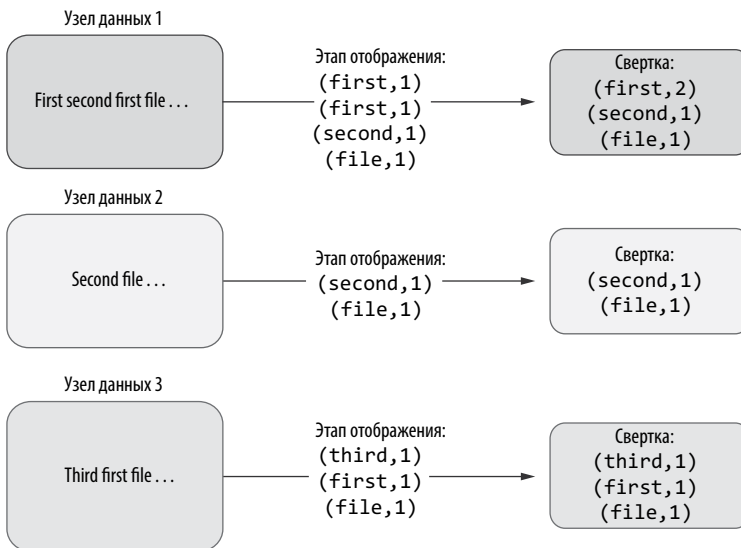


Рис. 8.14. Подсчет слов: первый этап на локальном компьютере, использующем локальность данных

В этом сценарии задействованы три узла данных. В левой части рис. 8.14 на каждом из узлов хранится большой файл данных с текстом. Каждый файл настолько велик, что отправить все файлы на один узел и провести вычисления там невозможно. Сначала текстовый файл разбивается на N слов. Для каждого слова на этапе отображения создается пара «ключ-значение»: ключом является слово, а значением — количество его вхождений. На первом этапе обработки значение всегда равно 1.

На первый взгляд этот этап кажется наивным и лишним. Однако здесь важно создание ключа секционирования для каждой записи. *Ключ секционирования* — это просто слово, для которого выполняется подсчет слов. Все пары с одним ключом секционирования в конечном итоге отправляются на один узел данных. Вскоре мы к этому вернемся.

После того как данные секционированы, можно выполнить локальную свертку. Это означает, что все пары с одним ключом (слово) сливаются и сворачиваются в новую пару, у которой ключ остается прежним, но счетчик изменяется для некоторых слов. Операция свертки на узле данных 1 сворачивает два вхождения пары $(first, 1)$ в одно вхождение $(first, 2)$. Это первый этап подсчета слов, который выполняется локально на каждом узле. После локальной свертки всех пар все готово ко второму этапу.

На втором этапе (рис. 8.15) данные перемещаются по сети (перетасовка данных). В соответствии с алгоритмом секционирования данные распределяются по N узлам. Здесь важно, что данные с одним ключом секционирования (слово) всегда попадают на один узел данных.

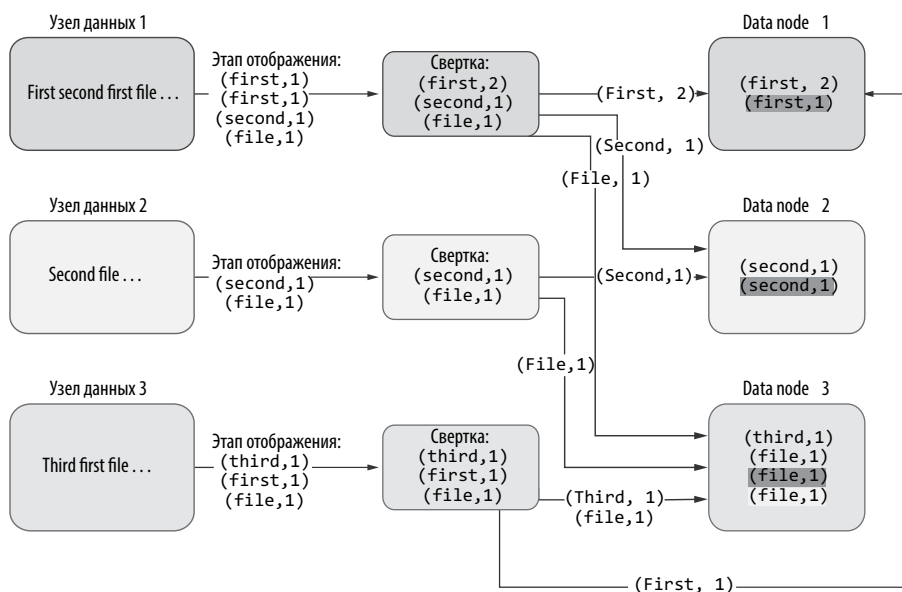


Рис. 8.15. Подсчет слов: второй этап с перетасовкой данных

Видим, что первый узел данных обрабатывает пары для ключа секционирования, указанного первым: все пары с этим ключом секционирования отправляются на этот узел данных. Пара `(first, 2)` остается на том же узле данных, так что для нее перетасовка данных не требуется. Однако третий узел содержит пару `(first, 1)` для отправки по сети, которая должна быть передана на первый узел данных. После того как данные перемещены, можно переходить к последнему этапу свертки.

Очень важно, что на этом этапе обработки мы можем не сомневаться, что все данные для одного ключа секционирования находятся на одном узле данных. Следовательно, можно выполнить другую свертку, использующую локальность данных. Операции свертки могут выполняться параллельно для разных ключей секционирования, что ускоряет вычисления.

Наконец, как показано на рис. 8.16, в результате обработки создается одна пара для каждого проанализированного слова. Результат может быть сохранен в файловой системе, базе данных, очереди и т. д. для дальнейшей обработки.

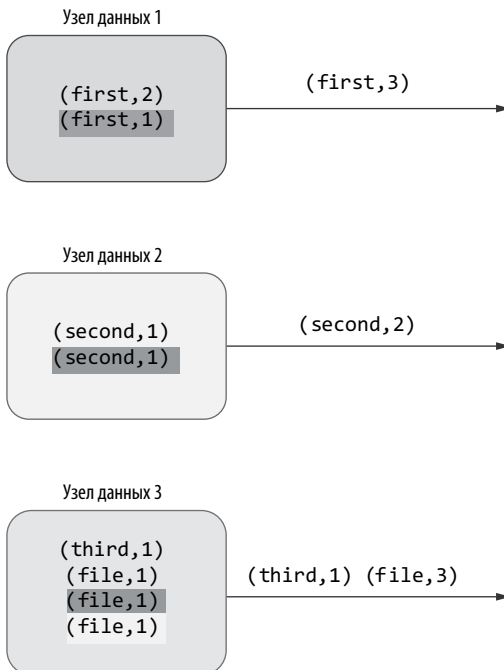


Рис. 8.16. Подсчет слов: последний этап с итоговой сверткой

Важно отметить, что в случае расфазировки данных в этом решении возникнут проблемы. Допустим, имеется один ключ секционирования, содержащий большую часть данных, которые необходимо обработать. Тогда все данные будут

отправлены на один узел. Если данные не поместятся на диске/в памяти этого узла, провести свертку этих данных для этого конкретного ключа секционирования не удастся. И снова секционирование данных и их распределение имеют очень большое значение.

Решение MapReduce довольно сложное по сравнению со сценариями с одним узлом. Тем не менее при выполнении обработки в контексте больших данных (то есть когда все данные невозможно разместить на одном узле) приходится мириться с этими трудностями, если мы хотим решить свои задачи.

Вычислим, насколько медленно дисковое решение MapReduce по сравнению с обработкой, основанной на работе с памятью. Сравним также MapReduce с загрузкой данных по сети и посмотрим, почему локальность данных так важна.

8.4.3. Вычисление времени обращения

Допустим, требуется обработать 100 Гбайт данных, разбитых на 1000 файлов. Рассчитаем время, необходимое для передачи 100 Гбайт данных по сети (перетасовка данных). Затем проведем вычисления для чтения данных с диска (как HDD, так и SSD). В завершение сравним результаты с ситуацией, в которой данные находятся в памяти.

Можно предположить, что обращения к памяти будут самыми быстрыми (SSD работает немного медленнее, а скорость HDD — на порядки ниже). Если вы не хотите углубляться в вычисления, можете смело пропустить этот раздел.

Для вычислений будут использоваться проверенные данные (<http://mng.bz/GGov>). Возможно, они немного устарели, но порядок соотношения этих чисел все еще актуален. Если данные находятся в оперативной памяти, обращение к ним происходит очень быстро: так, последовательное чтение 1 Мбайт из памяти занимает 250 000 нс (250 мкс).

Помните, что для сценария с использованием памяти данные необходимо загрузить заранее, но только один раз. По сравнению с диском (как SSD, так и HDD) различия велики — последовательное чтение 1 Мбайт с SSD: 1 000 000 нс (1000 мкс) (1 мс ~ 1 Гбайт÷с SSD, память 4X). Последовательное чтение 1 Мбайт с диска: 20 000 000 нс (20 000 мкс) (20 мс 80x memory, 20X SSD).

Наконец, чтение из сети, включая запрос и ответ, выглядит так: отправка пакета Калифорния -> Нидерланды -> Калифорния: 150 000 000 нс (150 000 мкс, 150 мс).

Конечно, наш вычислительный центр не будет пересылать данные между континентами при обработке больших данных. Тем не менее даже в локальном центре

передача данных по сети будет в несколько раз медленнее, чем обращение к локальным данным с диска. Для сетевых данных различия будут колоссальными.

Вычислим общее время, которое при обработке больших данных нужно выделить на загрузку данных в зависимости от места их хранения. Обработка 100 Гбайт данных, которые уже находятся в оперативной памяти, занимает: $250\,000\text{ нс} \times 1000\text{ (Мбайт)} \times 100\text{ (Гбайт)} = 25\,000\,000\,000\text{ нс} = 25\text{ с}$.

Для SSD это $1\,000\,000\text{ нс} \times 1000\text{ (Мбайт)} \times 100\text{ (Гбайт)} = 100\,000\,000\,000\text{ нс} = 100\text{ с}$. И наконец, для HDD оно составит $20\,000\,000\text{ нс} \times 1000\text{ (Мбайт)} \times 100\text{ (Гбайт)} = 2\,000\,000\,000\,000\text{ нс} = 2000\text{ с} = \sim 33\text{ мин}$.

Как видите, даже если использовать SSD для всех больших данных, их загрузка будет происходить в четыре раза медленнее, чем при работе с памятью. На практике, когда требуется хранить терабайты данных, они находятся на стандартных HDD, потому что это выгоднее. В таком случае обработка на базе HDD замедляется в 80 раз! На момент написания книги гигабайт памяти HDD стоил \$0,05, тогда как для SSD цена вдвое выше: \$0,10. Делаем вывод, что хранение данных на SSD обходится на 100 % дороже, чем на HDD. Результаты приведены в табл. 8.1.

Мы видели, что обработка больших данных Hadoop базируется на доступе к диску. Из-за этой медлительности и удешевления памяти сейчас в основном используется новый подход на основе доступа к памяти. Далее мы рассмотрим его подробнее.

Таблица 8.1. Время чтения с диска и из памяти

Тип ресурса	Размер (Гбайт)	Время (секунды)	Время (минуты)
ОЗУ	100	25	0,25
Диск SSD	100	100	~1,66
Диск HDD	100	2500	~33

8.4.4. Обработка данных в памяти

По мере снижения стоимости памяти архитектура новых средств обработки больших данных начала изменяться, чтобы в полной мере использовать память. Нередко можно увидеть кластеры вычислительных узлов с терабайтами оперативной памяти. Это позволяет инженерам создавать конвейеры данных, которые загружают максимально возможный объем данных в память. Здесь их обработка выполняется значительно быстрее, чем на диске. Один из самых известных и проверенных на практике фреймворков обработки больших данных — Apache Spark — использует память как основную точку интеграции между этапами обработки (рис. 8.17).

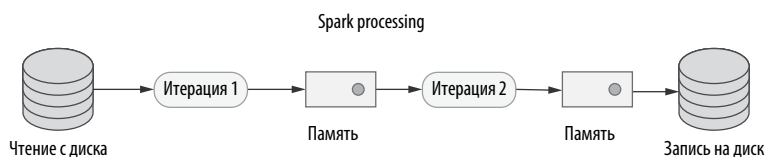


Рис. 8.17. Обработка больших данных в памяти в Apache Spark

Точка входа для обработки больших данных требует загрузки данных из файловой системы. Локальность данных требует загрузки данных с локального диска. При обработке в памяти данные загружаются в память компьютера. Когда текущий этап обработки завершается, результаты не записываются на диск (в отличие от обработки больших данных на базе Hadoop). Данные остаются в памяти узла, выполняющего обработку. На следующем этапе (преобразование, соединение) уже не нужно заново загружать данные с диска. Таким образом, затраты на загрузку данных с диска снимаются на всех этапах, кроме первого, когда данные загружаются. Результаты последнего этапа можно сохранить на диске.

Расчеты времени с диском и памятью показывают, что даже в лучшем случае (с использованием SSD) обработка на базе Hadoop в четыре раза медленнее. При обработке на базе Spark эти затраты приходится нести дважды — при загрузке данных и при сохранении результатов на диске. Та же схема обработки, при которой нужны только два перебора данных (преобразование), требует двух операций чтения с диска и двух операций записи на диск. В лучшем случае с использованием SSD обработка на базе Hadoop выполняется в четыре раза медленнее обработки на базе Spark: Hadoop — $100 \text{ с} \times 2 \text{ чтения} + 100 \text{ с} \times 2 \text{ записи} = 400 \text{ с}$, Spark — $25 \text{ с} \times 1 \text{ чтение} + 25 \text{ с} \times 1 \text{ запись} = 50 \text{ с}$. В итоге $400 \div 50 = 8$.

На практике запись на диск часто медленнее чтения с диска. Из-за этого различия между Spark и Hadoop становятся еще заметнее. Кроме того, реальные конвейеры больших данных обычно не ограничиваются только двумя этапами (преобразования). Некоторые конвейеры включают десять и более этапов до получения конечного результата. Для таких конвейеров больших данных необходимо умножить вычисления на количество этапов.

Очевидно, чем больше этапов, тем более заметны различия между обработкой данных, хранящихся в памяти и на диске. Наконец, как уже говорилось, диски HDD все еще используются из-за их экономичности. Вычислим общее время для обработки данных на диске на базе Hadoop с HDD: $33 \text{ мин} \times 2 \text{ чтения} + 33 \text{ мин} \times 2 \text{ записи} = 132 \text{ мин} = 2 \text{ часа и } 12 \text{ мин}$. Различия между обработкой данных в памяти и на диске колоссальны: 50 с против более 2 часов!

Надеюсь, эти числа убедят вас, что при создании современных конвейеров обработки больших данных следует стараться строить их на базе инструментов, использующих память, таких как Apache Spark. В следующем разделе мы реализуем соединение с использованием Apache Spark.

8.5. РЕАЛИЗАЦИЯ СОЕДИНЕНИЙ С ИСПОЛЬЗОВАНИЕМ АРАСНЕ SPARK

Прежде чем браться за реализацию логики, стоит вспомнить основы Apache Spark. Это библиотека на базе Scala для обработки больших данных, которая позволяет хранить промежуточные результаты в памяти. Как уже говорилось, иногда хранить все данные в оперативной памяти невозможно. Spark позволяет указать, что нужно делать в подобных ситуациях.

Spark также предоставляет настройку `StorageLevel`, которая позволяет указать, должны ли данные храниться только в памяти или могут сбрасываться на диск при ее заполнении. Если выбрать первый вариант, в процессе произойдет сбой, указывающий на нехватку памяти. Можно применить разбиение данных, чтобы они уместились в памяти компьютера. Если выбрать второй вариант, то данные будут сохранены на диске без сбоя процесса. Обработка завершится, но потребует гораздо больше времени. Как видите, Spark позволяет создавать обработку на базе памяти. Но что с локальностью данных?

Чтобы понять, как добиться локальности данных при использовании Spark, необходимо понимать архитектуру системы (рис. 8.18).

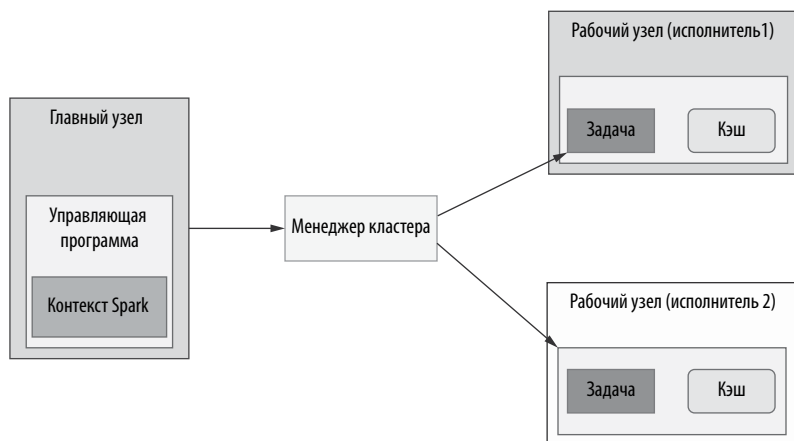


Рис. 8.18. Архитектура Spark

Spark использует архитектуру «главный узел/рабочие узлы». Каждый отдельный процесс Spark работает на узлах, содержащих обрабатываемые данные. Допустим, у вас три узла данных: один главный узел Spark и два исполнителя. Главный узел Spark представляет собой специальный процесс, отвечающий за обработку и отправку вычислений к данным.

Программа, которую мы напишем в этом разделе, использует Spark и отправляется на главный узел. Он сериализует программу (по аналогии с тем, как

это делалось в первом разделе) и отправляет ее узлам-исполнителям, которые выполняются на узлах данных, содержащих обрабатываемые данные. После этого можно построить обработку, использующую локальность данных. В этом случае исполнитель обрабатывает свои локальные данные в секциях, хранящихся на этом узле. Второй исполнитель обрабатывает секции, хранящиеся на втором узле.

Если связать эту схему с нашим примером с данными пользователей и кликов, исполнитель обработает данные некоторой части пользователей. Второй исполнитель находится на узле, на котором хранятся данные остальных пользователей. Кэш на исполнителях Spark — оперативная память. Обрабатываемые данные загружаются с диска или по сети и сохраняются в памяти.

Меньший набор данных о кликах `clicks`, участвующий в процессе соединения, отправляется всем исполнителям. Управляющий компонент на главном узле получает данные о кликах от узла, содержащего эти данные. Важно заметить, что управляющий процесс должен располагать достаточным объемом памяти для хранения данных. Затем данные кликов отправляются всем исполнителям, где они сохраняются в кэше (ОЗУ). Таким образом, память, доступная этим процессам, также должна быть достаточно большой для хранения данных.

8.5.1. Реализация соединения без рассылки

Начнем со сценария использования, в котором соединяются данные пользователей и кликов, но не будем делать утверждений относительно размера обоих наборов данных. Сначала введем простую операцию соединения без оптимизации. Затем проанализируем план выполнения, который покажет, как ядро Spark интерпретирует и выполняет операцию.

Примеры кода в этом разделе написаны на Scala, потому что этот язык позволяет вести динамичную и удобочитаемую обработку больших данных. Кроме того, Scala — родной язык Spark. В следующем листинге приведена простая модель данных для нашего примера.

Листинг 8.1. Модель данных

```
case class UserData(userId: String, data: String)

case class Click(userId: String, url: String)
```

Оба набора данных содержат идентификатор пользователя, который нужен для соединения данных. У пользователя с ним связано поле данных. Клик выполняется в контексте конкретного URL, поэтому это поле присутствует в модели данных.

В нашем примере мы используем Spark Dataset API (<http://mng.bz/zQKB>), позволяющий применять синтаксис, сходный с SQL. Это API более высокого уровня, инкапсулирующий RDD (<http://mng.bz/0wpN>).

Для тестирования данных будут задействованы имитации данных пользователей и кликов. В реальных приложениях данные будут загружаться из файловой системы при помощи объектов чтения данных. В следующем листинге показано чтение данных Avro (<http://mng.bz/KBQj>) из каталога HDFS.

Листинг 8.2. Чтение данных Avro

```
val usersDF = spark.read.format("avro").load("users/2020/10/10/users.avro")
```

Для простоты определим имитации двух наборов данных. В следующем листинге показано, как это делается.

Листинг 8.3. Имитация наборов данных пользователей и данных о кликах

```
import spark.sqlContext.implicits._
val userData =
  spark.sparkContext.makeRDD(List(
    UserData("a", "1"),
    UserData("b", "2"),
    UserData("d", "200")
  )).toDS()

val clicks =
  spark.sparkContext.makeRDD(List(
    Click("a", "www.page1"),
    Click("b", "www.page2"),
    Click("c", "www.page3")
  )).toDS()
```

Здесь набор `userData` заполняется строками данных с идентификаторами `a`, `b` и `d`. Наконец, RDD преобразуется в `Dataset` функцией `toDS()`, как показано в листинге 8.3. Мы хотим работать с набором данных, потому что он предоставляет улучшенный API и оптимизации поверх RDD API.

Логика соединения проста, но она скрывает значительную часть информации. В следующем листинге данные пользователей соединяются с данными о кликах.

Листинг 8.4. Соединение без утверждений

```
val res: Dataset[(UserData, Click)]
  = userData.joinWith(clicks, userData("userId") === clicks("userId"), "inner")
```

Здесь данные `userData` соединяются с `clicks`. При этом выполняется внутреннее соединение по полю `userId` обоих наборов данных. В результате при выполнении запроса будут получены два результата, как показывает следующий листинг.

Листинг 8.5. Два результата при внутреннем соединении

```
res.show()
assert(res.count() == 2)
+-----+-----+
|   _1|           _2|
+-----+-----+
|[b,2]| [b,www.page2]|
|[a,1]| [a,www.page1]|
+-----+-----+
```

Результат представлен в виде таблицы. В левом столбце содержатся данные пользователя, а в правом — данные о кликах. Данные пользователя `userId d` не включены, так как для него нет соответствующих данных о кликах. То же происходит с данными о кликах для пользователя `userId c`.

Соединение скрывает значительную сложность. Чтобы оценить ее, можно извлечь реальный физический план запроса. Он показывает, какая стратегия соединения была выбрана. Для извлечения физического плана выполните метод `explain()`, как показано в следующем листинге. Метод возвращает подробный физический план.

Листинг 8.6. Получение физического плана запроса

```
res.explain()
== Physical Plan ==
*SortMergeJoin [_1#10.userId], [_2#11.userId], Inner
:- *Sort [_1#10.userId ASC], false, 0
: +- Exchange hashpartitioning(_1#10.userId, 200)
:    +- *Project [struct(userId#2, data#3) AS _1#10]
:       +- Scan ExistingRDD[userId#2,data#3]
+- *Sort [_2#11.userId ASC], false, 0
   +- Exchange hashpartitioning(_2#11.userId, 200)
      +- *Project [struct(userId#7, url#8) AS _2#11]
         +- Scan ExistingRDD[userId#7,url#8]
```

Как видно из описания, оба набора данных обрабатываются по одной схеме. Сначала они сортируются по возрастанию. Когда набор отсортирован, к нему применяется алгоритм хеш-секционирования. Никаких утверждений о данных нет. Этот план соответствует сценарию использования, требующему перетасовки данных. Один из наборов необходимо передать исполнителю, содержащему другие части данных.

Так как данные отсортированы, ядро запросов Spark может применить оптимизации — например, перемещение только некоторого диапазона данных. Этот подход применяется обоснованно и иногда дает лучшие результаты, чем оптимизации, назначаемые для ядра запросов. Тем не менее важно оценить созданное решение и сравнить его с другим. Может оказаться, что созданные вручную кустарные оптимизации уступают стандартной логике оптимизатора запросов Spark. Теперь посмотрим на план соединения, который использует метод широковебчательной рассылки, описанный в разделе 8.3.3.

8.5.2. Реализация соединения с рассылкой

На следующем шаге реализуем поведение соединения, при котором один из наборов данных (`clicks` в нашем случае) рассылается по всем узлам данных. Для этого необходимо изменить логику соединения, упаковав рассылаемый набор данных в функции `broadcast()`. Используем для этого набор данных `clicks`. Полный набор тестов приведен в следующем листинге.

Листинг 8.7. Соединение с рассылкой

```
test("Should inner join two DS whereas one of them is broadcast") {
  import spark.sqlContext.implicits._
  val userData =
    spark.sparkContext.makeRDD(List(
      UserData("a", "1"),
      UserData("b", "2"),
      UserData("d", "200")
    )).toDS()

  val clicks =
    spark.sparkContext.makeRDD(List(
      Click("a", "www.page1"),
      Click("b", "www.page2"),
      Click("c", "www.page3")
    )).toDS()

  //Если
  val res: Dataset[(UserData, Click)]
    = userData.joinWith(broadcast(clicks), userData("userId") ===
clicks("userId"), "inner")

  //To
  res.explain()
  res.show()
  assert(res.count() == 2)
```

Запрос возвращает те же данные, что и предыдущий, потому что логика не изменилась. Для нас интерес представляет физический план запроса, приведенный в следующем листинге.

Листинг 8.8. Просмотр физического плана запроса с рассылкой

```
* == Physical Plan ==
* *BroadcastHashJoin [_1#234.userId], [_2#235.userId], Inner, BuildRight
* :- *Project [struct(userId#225, data#226) AS _1#234]
* : +- Scan ExistingRDD[userId#225,data#226]
* +- BroadcastExchange HashedRelationBroadcastMode(List(input[0,
struct<userId:string,url:string>, false].userId))
* +- *Project [struct(userId#230, url#231) AS _2#235]
* +- Scan ExistingRDD[userId#230,url#231]
```

Вы заметили, что физический план существенно изменился? Во-первых, данные уже не отсортированы. Исполнительное ядро Spark удалило этот шаг, потому что отправлять части одного из наборов данных не требуется; следовательно, его не нужно разбивать. Шаг **Broadcast-Exchange** отвечает за отправку данных **clicks** всем узлам данных. Когда эти данные находятся на всех узлах, Spark выполняет шаг **Scan**, использующий результат хеширования для нахождения подходящих данных.

Получение результата еще не все. В реальном проекте следует измерить время выполнения обоих решений. Как уже говорилось, может оказаться, что стандартное ядро запросов Spark работает более эффективно.

При рассылке данных по узлам необходима абсолютная уверенность, что данные поместятся в памяти машины. Если рассылаемые данные начинают неконтролируемо увеличиваться в размерах, следует серьезно пересмотреть стратегию рассылки. В следующей главе будут рассмотрены стратегии выбора сторонних библиотек, используемых в коде.

ИТОГИ

- Перемещение данных к вычислениям — более простой, но затратный метод. Для крупных наборов данных он не подходит, так как требует пересылки слишком больших объемов данных по сети.
- Локальность данных в полной мере используется при отправке вычислений к данным. Этот метод сложнее, но он оправдывает усилия для больших данных. С ним не приходится перемещать столько данных, что существенно упрощает обработку.
- Обработка, использующая локальность данных, проще в распараллеливании и масштабировании, чем обработка без нее.
- В экосистеме больших данных информация распределяется по нескольким компьютерам посредством секционирования.
- Автономное и сетевое секционирование обладают разными характеристиками. Сетевое секционирование позволяет выполнять оптимизацию для паттернов запросов, а автономное более универсально, потому что паттерны обращений к данным часто неизвестны заранее.
- Автономное секционирование, основанное на датах, часто применяется на практике и обеспечивает большую гибкость.
- Некоторые типы соединений могут в полной мере использовать локальность данных при выполнении соединения на одной физической машине. Другие типы соединений, для которых нужны более широкие данные, требуют перетасовки данных.

- Чтобы сократить перетасовку данных, следует уменьшить количество секций, необходимых для операций соединения.
- Утверждения о данных позволят использовать стратегию соединения с рассылкой.
- Обработка больших данных, хранящихся на диске, более детальна, но по скорости уступает обработке данных в памяти. Hadoop реализует первую стратегию, в Spark используется вторая.
- Для реализации соединений можно использовать Apache Spark API.
- Анализ физических планов позволяет анализировать запросы. Например, можно воспользоваться методом рассылки, а затем посмотреть, как он используется ядром выполнения запросов.
- Необходимо знать данные, чтобы анализировать достоинства и недостатки разных стратегий соединения.

Сторонние библиотеки: используемые библиотеки становятся кодом

https://t.me/it_boooks

В этой главе:

- ✓ Ответственность за импортируемые библиотеки.
- ✓ Анализ сторонних библиотек на удобство тестирования, стабильность и масштабируемость.
- ✓ Принятие решений о повторной реализации логики и импортирование кода, который вам не принадлежит.

Программные системы строятся в условиях ограничений времени и бюджета. Из-за этого практически невозможно написать весь программный код самостоятельно. Почти каждое приложение взаимодействует с операционной и файловой системами, а также внешними средствами ввода/вывода. Обычно никто не реализует эту логику самостоятельно. Программисты используют существующие библиотеки, которые предоставляют подобную функциональность. Такие библиотеки называются *сторонними*, потому что они созданы не командой или компанией разработчиков продукта. Они могут разрабатываться сообществом открытого кода или другими компаниями, специализирующимися на конкретной части архитектуры системы. Например, при отправке данных внешней системе HTTP часто выбирается готовая реализация клиента HTTP.

Когда вы выбираете существующую стороннюю библиотеку и используете ее в своей кодовой базе, вы берете на себя всю ответственность за этот код, хотя его

вы не разрабатывали и не публиковали. Конечного пользователя не интересует, какую из библиотек вы выбрали. Они не знают, реализована ли конкретная часть кода вами или кем-то еще. Пока система работает как надо, проблем нет. Но когда происходит сбой, пользователи это заметят. Может оказаться, что сбой был вызван ошибкой в стороннем продукте. Это означает, что сторонний код был недостаточно хорошо протестирован или в отношении него имелись ложные предположения.

В этой главе вы узнаете, как выбрать для приложения надежную стороннюю библиотеку. Вы узнаете о самых частых ошибках и научитесь проверять предположения о коде, который вам не принадлежит.

9.1. ИМПОРТИРОВАНИЕ БИБЛИОТЕКИ И ОТВЕТСТВЕННОСТЬ ЗА ЕЕ НАСТРОЙКИ: БЕРЕГИТЕСЬ ЗНАЧЕНИЙ ПО УМОЛЧАНИЮ

В некоторых библиотеках и фреймворках — таких, как Spring (<https://spring.io/>), — приоритет отдается соглашениям, а не конфигурации. Такой подход позволяет потенциальным клиентам использовать конкретную библиотеку немедленно, без необходимости настраивать конфигурацию. Явные настройки заменяются простотой UX. Пока специалисты знают об этом компромиссе и его ограничениях, реальной опасности нет.

Использование программных компонентов, не требующих значительной предварительной настройки конфигурации, существенно упрощает и ускоряет прототипирование и эксперименты. Эти фреймворки строятся с применением оптимальных практик и паттернов, они хороши и достаточны — если помнить об их недостатках и проблемах.

ПРИМЕЧАНИЕ

Концепции фреймворков и библиотек часто используются как синонимы. Фреймворк предоставляет каркас для построения приложений, но настоящая логика реализуется в приложении. Логике необходимо каким-то образом предоставить фреймворку: посредством наследования, композиции, прослушивателей и т. д. (например, во фреймворках с внедрением зависимостей). С другой стороны, библиотека уже реализует некоторую логику, которую можно только вызывать из кода. Например, библиотека клиента HTTP предоставляет средства для вызова сервисов HTTP.

В том, что большая часть конфигурации основана на соглашениях, тоже есть минусы. При использовании сторонней библиотеки появляется искушение не зарываться слишком глубоко в настройки ее конфигурации. Если оставить дефолтные значения не заданными, придется положиться на конфигурацию, поставляемую

вместе с библиотекой. Такие значения обычно выбираются согласно логике, подкрепленной исследованиями. Но даже если дефолтные значения выбраны грамотно, они могут оказаться неподходящими для конкретной ситуации.

Рассмотрим простой сценарий с использованием сторонней библиотеки, ответственной за вызовы HTTP. В качестве примера возьмем библиотеку OkHttp (<https://square.github.io/okhttp/>). Мы хотим запросить данные, доступные в конечной точке сервиса `/data`. Для тестирования имитируем конечную точку HTTP с использованием библиотеки WireMock (<http://wiremock.org/>). Для конечной точки `/data` будет создана заглушка, которая возвращает код статуса OK и данные тела некоторой сущности. Этот код представлен в следующем листинге.

Листинг 9.1. Имитация сервиса HTTP

```
private static WireMockServer wireMockServer;
private static final int PORT = 9999;
private static String HOST;

@BeforeAll
public static void setup() {
    wireMockServer = new WireMockServer(options().port(PORT));
    wireMockServer.start();
    HOST = String.format("http://localhost:%s", PORT);
    wireMockServer.stubFor(
        get(urlEqualTo("/data"))
            .willReturn(aResponse()
                .withStatus(200)
                .withBody("some-data")));
}
```

Запускает сервер WireMock на выделенном порте PORT

Сохраняет местоположение в переменной HOST

Имитирует ответ HTTP с кодом статуса 200 и некоторыми данными

Логика клиента OkHttp для отправки запроса сервису и получения ответа достаточно прямолинейна. В листинге 9.2 URL-адрес строится на базе переменной HOST. Затем создается клиент OkHttp с использованием строителя и выполняется вызов. Наконец, тест проверяет, что ответ содержит 200 и контент совпадает с тем, который был предоставлен WireMock.

Листинг 9.2. Построение клиента HTTP с настройками по умолчанию

```
@Test
public void shouldExecuteGetRequestsWithDefaults() throws IOException {
    Request request = new Request.Builder().url(HOST + "/data").build();

    OkHttpClient client = new OkHttpClient.Builder().build();
    Call call = client.newCall(request);
    Response response = call.execute();

    assertThat(response.code()).isEqualTo(200);
    assertThat(response.body().string()).isEqualTo("some-data");
}
```

Заметьте, что клиент HTTP создается как строитель, но без явно заданных настроек. Код выглядит просто, и можно быстро приступить к разработке. К сожалению, его нельзя использовать в финальной версии приложения в таком формате. Помните: импортированная сторонняя библиотека становится вашим собственным кодом. Так как этот раздел посвящен настройкам по умолчанию, разберемся, какие из них могут создать проблемы.

При анализе настроек сторонних библиотек необходимо понимать их главные конфигурации. В контексте каждого клиента HTTP исключительно важная роль принадлежит тайм-аутам. Они влияют на производительность и условия SLA вашего сервиса. Например, если SLA составляет 100 мс и вы обращаетесь с вызовом к другим сервисам для выполнения запроса, другой вызов должен завершиться быстрее времени SLA вашего сервиса. Выбор правильного тайм-аута исключительно важен, если вы хотите сохранить условия SLA.

Высокие тайм-ауты также опасны в архитектуре микросервисов. Для реализации бизнес-функций в этой архитектуре часто приходится выдавать несколько сетевых вызовов. Например, один микросервис может вызывать ряд других. Некоторые из них могут связываться с микросервисами следующего уровня и т. д. Если в таком сценарии один из сервисов зависает в ходе обработки запроса, это может породить каскадные сбои в других сервисах, которые его вызывают. Чем выше тайм-аут, тем больше времени займет обработка одного запроса и тем больше вероятность каскадных сбоев. Такие сбои могут быть хуже нарушения SLA, потому что они создают риск того, что в системе произойдет критическая ошибка и она перестанет работать.

Посмотрим, как поведет себя клиент при слишком долгом запросе к конечной точке. Протестируем его в течение 5 с (5000 мс). Для моделирования такого сценария в WireMock можно воспользоваться методом `withFixedDelay()` (см. следующий листинг).

Листинг 9.3. Эмуляция медленной конечной точки

```
wireMockServer.stubFor(  
    get(urlEqualTo("/slow-data"))  
        .willReturn(aResponse()  
            .withStatus(200)  
            .withBody("some-data")  
            .withFixedDelay(5000));
```

Для обращения к новой конечной точке можно воспользоваться URL-адресом `/slow-data`. Запрос выполняется с использованием той же логики, но мы будем измерять время, необходимое для выполнения запросов HTTP, как показано в листинге ниже.

Листинг 9.4. Измерение времени запроса клиента HTTP

```

Request request = new Request.Builder()
    ➔ .url(HOST + "/slow-data").build();
OkHttpClient client = new OkHttpClient.Builder().build();
Call call = client.newCall(request);

long start = System.currentTimeMillis();
Response response = call.execute();
long totalTime = System.currentTimeMillis() - start;

assertThat(totalTime).isGreaterThanOrEqualTo(5000);
assertThat(response.code()).isEqualTo(200);
assertThat(response.body().string()).isEqualTo("some-data");

```

Выполняет запрос к конечной точке /slow-data

Измеряет общее время выполнения

Проверяет, что запрос занял не менее 5000 мс

Заметили, что выполнение запроса заняло не менее 5000 мс? Это произошло из-за того, что сервер HTTP WireMock ввел указанную задержку. Если наш код, который должен выполнять запрос за 100 мс, обратится с вызовом к этой конечной точке, вызов будет достаточно медленным для нарушения SLA.

Вместо получения ответа в течение 100 мс (независимо от того, содержит ли он признак успеха или неудачи) клиенты будут блокироваться в ожидании на 5000 мс. Также это значит, что поток, выполняющий эти запросы, может блокироваться на соответствующее время. Поток, который должен выполнить ~50 запросов ($5000 \div 100$ мс), остается заблокированным; в это время он не может обрабатывать другие запросы, что влияет на общую производительность сервиса. Проблема может не возникнуть, если слишком долго ожидает только один поток. Но если все назначенные потоки (или хотя бы большинство) будут блокироваться на длительное время, вы начнете замечать проблемы с производительностью.

Как оказалось, эта ситуация возникает из-за настроек тайм-аута по умолчанию. Клиент сообщает о сбое обработки запроса, если ожидание превышает SLA сервиса (100 мс). Если запрос завершается неудачей, клиент может повторно выдать его вместо того, чтобы ждать ответа в течение 5000 мс. Взглянув на тайм-аут чтения OkHttpClient (<http://mng.bz/9KP7>), вы заметите, что по умолчанию он составляет 10 с!

ПРИМЕЧАНИЕ

Проверка дефолтных настроек важна не только для сторонних библиотек, но и для стандартных наборов средств разработки (SDK). Например, при использовании HttpClient, поставляемого с Java JDK (<http://mng.bz/jylr>), по умолчанию выбирается бесконечный тайм-аут!

Это означает, что каждый запрос HTTP может блокировать выполнение вызывающей стороны на время до 10 с. Такая ситуация далека от

идеала. В реальной системе тайм-ауты следовало бы настроить в соответствии с SLA.

Предполагается, что код должен выполнить запрос к конечной точке `slow-data` за время 100 мс. Также предполагается, что вызываемый сервис имеет определенный SLA в 99-м процентиле, равный 100 мс. Это означает, что 99 из 100 запросов будут выполняться в пределах 100 мс. Могут присутствовать отдельные выбросы, занимающие большее время. Можно смоделировать такой выброс, для выполнения которого требуется 5000 мс.

Снова выполним запрос HTTP, но на этот раз зададим тайм-аут явно, вместо того, чтобы полагаться на значение по умолчанию. Обратите внимание на метод `readTimeout()` в следующем листинге — он используется для назначения тайм-аута.

Листинг 9.5. Выполнение запроса HTTP с явным назначением тайм-аута

```
@Test
public void shouldFailRequestAfterTimeout() {
    Request request = new Request.Builder().url(HOST + "/slow-data").build();

    OkHttpClient client = new OkHttpClient
        .Builder()
        .readTimeout(Duration.ofMillis(100)).build();
    Call call = client.newCall(request);

    long start = System.currentTimeMillis();
    assertThatThrownBy(call::execute).isInstanceOf(SocketTimeoutException.class);
    long totalTime = System.currentTimeMillis() - start;

    assertThat(totalTime).isLessThan(5000);
}
```

Назначает тайм-аут чтения 100 мс

Сбой происходит быстрее, и запрос занимает менее 5000 мс

Вызов метода `execute` инициирует фактическое выполнение запроса HTTP. Запрос завершается по тайм-ауту приблизительно через 100 мс, потому что это значение было задано вызовом `readTimeout()`. По прошествии указанного времени на сторону вызова распространяется исключение. Таким образом, ошибка не повлияет на SLA сервиса. Затем делается повторная попытка выполнения запроса (если он идемпотентен), или же информация о сбое сохраняется на будущее. Что еще важнее, медленный ответ сервиса HTTP не блокирует поток на долгое время. Следовательно, это никак не повлияет на производительность сервиса.

При импорте любой сторонней библиотеки следует знать ее настройки и параметры. Неявные настройки могут подойти для построения прототипа, но явные и тщательно подобранные для контекста настройки обязательны для реальных систем. В следующем разделе будут рассмотрены модели параллельного выполнения и масштабируемость библиотек, которые могут использоваться в кодовой базе.

9.2. МОДЕЛИ ПАРАЛЛЕЛЬНОГО ВЫПОЛНЕНИЯ И МАСШТАБИРУЕМОСТЬ

Мы добавляем сторонние библиотеки в кодовую базу, чтобы они выполняли некоторую работу. Значит, нам необходимо вызвать API, дождаться выполнения и (возможно) получить результат. Эта простая схема скрывает часть сложности, относящейся к модели обработки. Когда вы вызываете код, который вам не принадлежит, следует обратить особое внимание на его модель параллельного выполнения.

Первый сценарий, который мы рассмотрим, будет довольно простым. Имеется программа, работающая по последовательной схеме с блокированием. Структура такой программы показана на рис. 9.1.

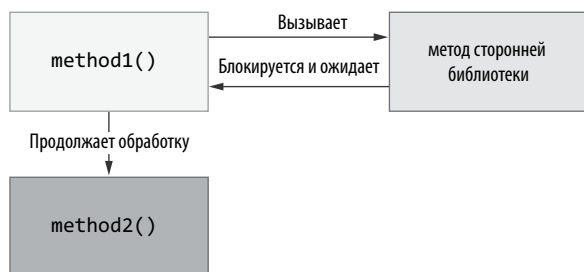


Рис. 9.1. Программа с блокирующим вызовом из кодовой базы

В нашей программе `method1()` выполняет метод сторонней библиотеки. Этот метод является блокирующим, то есть поток вызывающей стороны `method1()` блокируется до возвращения управления методом сторонней библиотеки. Когда метод вернет управление, выполнение вызывающей стороны продолжается и вызывается метод `method2()`.

Ситуация усложняется при использовании асинхронной, неблокирующей модели выполнения. У некоторых веб-фреймворков (таких как, Node.js, Netty, Vert.x и других) обработка базируется на модели цикла событий (рис. 9.2).

В таком контексте каждый запрос или часть работы, которая должна обрабатываться, помещается в очередь. Например, когда веб-сервер должен обработать запрос HTTP, рабочий поток, получающий запрос, не выполняет фактическую обработку. Он помещает данные, которые нужно обработать, в очередь. Затем поток из пула потоков, ответственного за обработку, берет данные этой очереди и выполняет фактическую обработку. Выполняя вызов любого метода из кода, который не может блокироваться, нужно внимательно подходить к вызову стороннего кода (рис. 9.3).

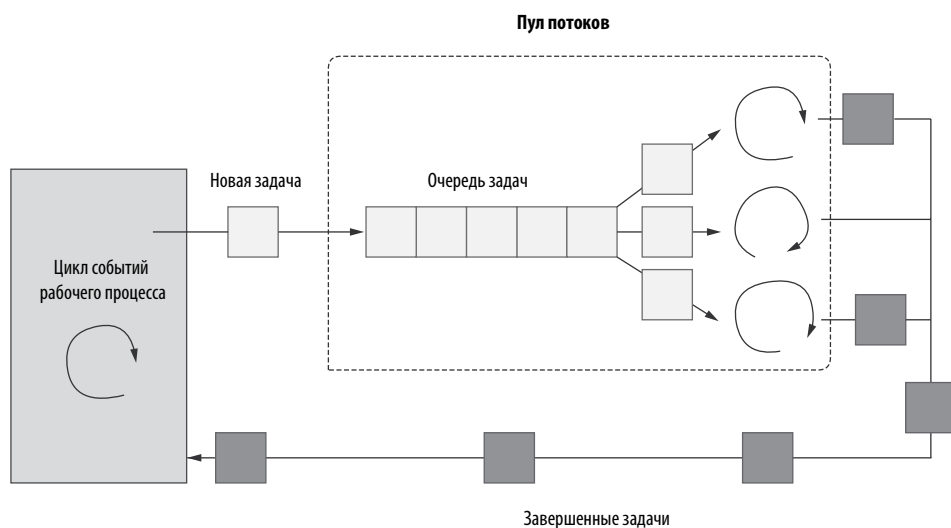


Рис. 9.2. Модель обработки с циклом событий



Рис. 9.3. Блокирующий вызов из неблокирующего кода

В таком сценарии все происходит в границах одного рабочего потока. Асинхронная логика получает данные и может выполнить некоторую предварительную обработку (например, десериализацию из байтовых данных). Затем информация данных помещается в очередь. Эта операция должна быть быстрой и неблокирующей, чтобы не тормозить обработку входящих запросов. Если вызвать из

такого кода сторонний код, возникает риск блокирования основной обработки. Таким образом, общее быстродействие приложения ухудшается.

По этим причинам необходимо знать, как выполняется вызываемый код. Блокируется ли он? Работает ли асинхронно или синхронно? Посмотрим, как использовать стороннюю библиотеку, которая предлагает обе модели.

9.2.1. Использование асинхронных и синхронных API

Допустим, вы интегрируетесь со сторонней библиотекой для сохранения и загрузки сущностей. API является блокирующим, а это значит, что он не должен вызываться из асинхронного кода. В следующем листинге приведен код такого сценария.

Листинг 9.6. Блокирующий API

```
public interface EntityService {
    Entity load();
    void save(Entity entity);
}
```

Вызывающий поток блокируется независимо от того, вызывает ли он метод `load` или `save`, но это создает проблемы и ограничивает возможности использования этого API. Например, будет трудно подключить блокирующую обработку к уже существующему асинхронному коду. Кроме того, потоковая модель приложения может не допускать никакой блокировки (например, при использовании Vert.x).

Что делать, если вы хотите воспользоваться сторонней библиотекой, даже если она блокирующая? Простейший и самый очевидный способ — создать обертку вокруг блокирующего кода, как показано в листинге 9.7. Обертка делегирует фактическую обработку внешней библиотеке и предоставляет методы, которые могут использоваться асинхронно. Оба метода могут вернуть сущность `CompletableFuture` — обещание (promise), которое будет выполнено в будущем. Асинхронный код, который не допускает блокирование, вызывает только неблокирующие версии этих методов.

Листинг 9.7. Асинхронная обертка для синхронного вызова

```
public CompletableFuture<Entity> load() {
    return CompletableFuture.supplyAsync(entityService::load, executor);
}

public CompletableFuture<Void> save(Entity entity) {
    return CompletableFuture.runAsync(() -> entityService.save(entity), executor);
}
```

Обратите внимание: метод `load()` возвращает обещание `Entity`, которое может быть выполнено в любой момент. Вызывающая сторона способна объединять

асинхронные операции в цепочку без блокирования потока вызывающей стороны.

На первый взгляд, решение кажется простым. Тем не менее упаковать блокирующий код в асинхронную обертку не всегда просто. Асинхронные действия должны выполняться в отдельном потоке. Для этого нужно создать специальный пул потоков, который будет использоваться этим кодом. Пул необходимо контролировать и оптимизировать. Следует выбрать правильное количество потоков и создать очередь входящих операций, как показано в следующем листинге.

Листинг 9.8. Создание исполнителя

```
public WrapIntoAsync(EntityService entityService) {
    this.entityService = entityService;
    executor = new ThreadPoolExecutor(1, 10, 100, TimeUnit.SECONDS, new
        LinkedBlockingDeque<>(100));
}
```

Получает corePoolSize, maximumPoolSize, keepAliveTimeout и очередь задач

Найти оптимальную конфигурацию для кода, который вам не принадлежит, может быть непросто. Необходимо знать предполагаемый трафик и провести тесты производительности. Кроме того, если библиотека написана с использованием блокирования, ее быстродействие может быть хуже, чем у кода, написанного по асинхронной модели. Упаковка блокирующего кода может только отложить проблему масштабирования, не решая ее.

Если производительность критична и не существует сторонней библиотеки, решающей задачу по асинхронному принципу, рассмотрите возможность самостоятельной реализации ее отдельных частей. Возьмем ситуацию, в которой вы выбираете внешнюю библиотеку, предоставляющую асинхронный API в исходном состоянии. Следующий листинг показывает, как будет выглядеть API сервиса.

Листинг 9.9. Создание асинхронного API

```
public interface EntityServiceAsync {
    CompletableFuture<Entity> load();

    CompletableFuture<Void> save(Entity entity);
}
```

Все методы компонента возвращают обещание, указывающее на то, что обработка выполняется асинхронно. Это означает, что внутренняя реализация библиотеки, с которой происходит интегрирование, написана асинхронно. При таком подходе не придется реализовывать уровень для перехода от синхронности к асинхронности. Часто это означает, что пул потоков, который мы используем для выполнения асинхронной задачи, инкапсулирован внутри библиотеки. Он может быть уже оптимизирован для большинства сценариев использования. Тем не менее, как говорилось в первой части этой главы, следует помнить о значениях по умолчанию.

Тот факт, что пул потоков инкапсулируется в библиотеке, не означает, что он не создает потоки. Код вызывается из приложения. Потоки, созданные во внутренней реализации для целей вызываемой библиотеки, все равно захватывают ресурсы в приложении. Если в приложении используется блокирующая синхронная схема выполнения, асинхронный код вызвать проще, чем при необходимости вызывать блокирующий код в асинхронной схеме выполнения.

Единственное, что нужно сделать, — получить значение из возвращаемого объекта `CompletableFuture`. Также необходимо учитывать, что вызов является блокирующим, и этому действию рекомендуется передать разумный тайм-аут. Но если в приложении уже применяется блокирование, это не создаст проблем. Данный подход продемонстрирован в следующем листинге.

Листинг 9.10. Переход от неблокирующей модели к блокирующей

```
public class AsyncToSync {
    private final EntityServiceAsync entityServiceAsync;

    public AsyncToSync(EntityServiceAsync entityServiceAsync) {
        this.entityServiceAsync = entityServiceAsync;
    }

    Entity load() throws InterruptedException, ExecutionException,
        TimeoutException {
        return entityServiceAsync.load().get(100, TimeUnit.MILLISECONDS);
    }
}
```

Преобразованный метод возвращает Entity

Асинхронный вызов блокируется с получением значения

При использовании библиотеки, предоставляющей асинхронный или синхронный API, зачастую лучше выбрать асинхронную версию. Даже если сейчас в приложении используется блокирование, возможно, вы захотите перевести его на асинхронную обработку, чтобы улучшить его масштабируемость и производительность.

Если вы уже используете библиотеку, которая предоставляет асинхронный API, вам проще перейти на новую модель выполнения. Но если вы работаете с библиотекой, написанной для блокирующего выполнения, переход уже не будет таким простым. Вам придется предоставить переходный уровень и управлять пулом потоков. Более того, код, который не был изначально написан как асинхронный, часто реализуется по-другому. Упаковка вызовов в API обещаний станет быстрым обходным решением.

Производительность библиотеки, которая изначально создавалась по модели асинхронного выполнения, часто оказывается выше, чем у блокирующей версии. В этом нетрудно убедиться, особенно если весь процесс обработки асинхронный. Посмотрим, как масштабируемость приложения может ограничиваться библиотекой, при написании которой не учитывалась возможность масштабирования.

9.2.2. Распределенная масштабируемость

Когда приложение выполняется в распределенной среде, очень важно понимать возможности масштабирования сторонних библиотек, которые вы собираетесь использовать. Рассмотрим библиотеку, предоставляющую средства планирования для приложения (что-то вроде заданий `cron`). Ее главная обязанность — проверить, должна ли задача выполняться, и запускать ее при достижении временного порога.

Сторонней библиотеке требуется уровень долгосрочного хранения данных для решения ее задач. С каждой задачей связаны дата и время ее выполнения. После выполнения задачи библиотека планирования обновляет ее статус: **Success**, **Failed** или **None**, если задача еще не была обработана. Библиотека планирования схематически изображена на рис. 9.4.

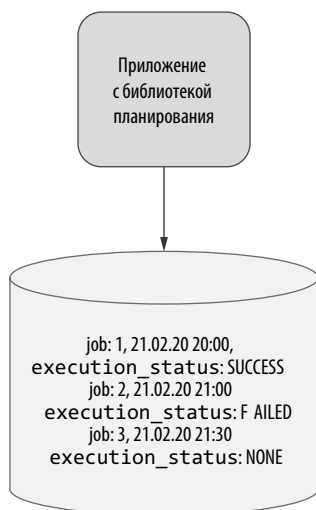


Рис. 9.4. Приложение с библиотекой планирования

Задачи, которые должны быть выполнены, хранятся в базе данных и загружаются приложением с некоторым интервалом. При разработке такой функциональности возникает искушение начать проектирование для одноузлового сценария использования. Интеграционные тесты могут проверить поведение библиотеки планирования со встроенной базой данных. Тем не менее при запросах из базы данных в такой ситуации проблемы могут быть не заметны.

Ситуация кардинально меняется при работе в распределенном контексте. Приложение, которому нужны средства планирования задач, можно развернуть на нескольких узлах, и задачи планирования не должны дублироваться. Это означает, что при развертывании приложения на нескольких узлах они должны

«договориться» о том, какой узел какую задачу будет обрабатывать. Иначе говоря, одна задача не может обрабатываться более чем одним узлом.

Такое требование означает, что состояние заданий должно либо глобально синхронизироваться, либо секционироваться. Если в выбранной библиотеке планирования не реализована логика масштабирования, появляется риск серьезных проблем с производительностью или даже правильностью работы. Будем считать, что приложение должно быть развернуто на трех узлах. Каждый из них использует библиотеку планирования, как показано на рис. 9.5.

Если библиотека планирования не реализована с учетом возможности масштабирования, все эти узлы будут конкурировать за запись задания в базе данных (рис. 9.5). Правильность таких изменений может достигаться за счет применения транзакций или глобальной блокировки конкретной записи. Тем не менее оба решения могут существенно повлиять на производительность библиотеки.

Эту проблему можно решить, если библиотека планирования поддерживает секционирование. Например, первый узел может отвечать за задачи в указанном диапазоне времени, другой узел — за задачи для другого периода времени и т. д. Здесь важно то, что в используемой библиотеке должна быть предусмотрена возможность масштабирования. Часто это нетривиальная задача, требующая тщательного планирования и технической проработки. Не все библиотеки проектируются для работы в таком контексте. Следовательно, при выборе библиотеки, которая заведомо будет выполняться в распределенной среде (на нескольких узлах), следует тщательно проанализировать ее и проверить ее работоспособность в распределенном контексте.



Рис. 9.5. Масштабирование библиотеки планирования по нескольким узлам

Другое решение для библиотеки планирования, которое можно применить, — архитектура «ведущий — ведомые». При таком подходе все запросы на планирование выполняются ведущим. Однако никакой реальной логики он не выполняет. Если на ведущем узле происходит сбой, один из ведомых узлов становится новым ведущим и начинает выполнять задания `cron`. Тем не менее библиотека планирования должна изначально проектироваться для работы в многоузловом контексте.

Понимание модели масштабируемости и необходимости использования глобального состояния позволит масштабировать приложение без проблем, которые трудно исправить. Если библиотека не проектировалась для работы в распределенной среде, могут возникнуть проблемы с масштабируемостью и работоспособностью. Они часто проявляются при развертывании приложений на N узлах, где N выше обычного количества узлов. Такие ситуации, как правило, возникают при выбросах трафика в приложении. Что еще хуже, обычно выбросы происходят, когда для продукта открываются хорошие бизнес-возможности. Например, на выходных. Это не самое подходящее время, чтобы узнать, что приложение зависит от библиотеки, которая не масштабируется.

И снова используемый вами код становится вашим кодом — так будут считать пользователи, которые заметят сбой. В следующем разделе рассматривается тестирование сторонних библиотек, призванное свести к минимуму такие проблемы.

9.3. ТЕСТИРУЕМОСТЬ

При выборе библиотеки с кодом, который вы не проектировали и не разрабатывали, не стоит доверять ей безраздельно. Не стоит делать практически никаких предположений. Тем не менее качество и правильность проверенных и популярных библиотек часто находятся на достаточно высоком уровне. В таких случаях тестирование с большой вероятностью проверяет предположения о библиотеке, а не правильность ее работы. Тестирование — лучший эксперимент и проверка пригодности сторонней библиотеки, которую вы собираетесь использовать в коде. Однако тестирование кода, на который невозможно повлиять, несколько отличается от тестирования собственного кода. Главная причина — этот код нелегко изменить (если вообще возможно).

Если вы хотите протестировать компонент из своей кодовой базы и выясняется, что код не позволяет повлиять на поведение, это относительно легко исправить. Например, если код инициализирует внутренний компонент, не давая вызывающей стороне возможности внедрить ложное или имитированное значение, можно провести рефакторинг кода без особых проблем. С другой стороны, при использовании сторонней библиотеки влиять на кодовую базу становится трудно или невозможно. Даже если вы примените изменения, время от изменения

до развертывания может быть значительным. Из-за этого, прежде чем выбрать стороннюю библиотеку, следует проверить ее на тестируемость.

Начнем с первой контрольной точки в списке оценки тестируемости. Она звучит так: предоставляет ли сторонняя библиотека тестовую библиотеку, которая позволяет ее протестировать?

9.3.1. Тестовая библиотека

При импортировании библиотеки, предоставляющей сложную функциональность, должна быть возможность относительно легко протестировать ее код, причем тестирование должно быть прямолинейным. Рассмотрим ситуацию, в которой требуется реализовать реактивную обработку. Для этого необходимо выбрать между парой библиотек, предоставляющих эту функциональность.

Начнем с реализации заготовки обработки, которая служит прототипом для более сложной логики (листинг 9.11). Требуется сложить все входящие числа в 10-секундном окне. Логика оперирует с потоком данных; это означает, что при поступлении событий они интерпретируются в контексте окна, после чего обработка продолжается.

Листинг 9.11. Реактивная обработка

```
public static Flux<Integer> sumElementsWithinTimeWindow(Flux<Integer> flux) {  
    return flux  
        .window(Duration.ofSeconds(10))  
        .flatMap(window -> window.reduce(Integer::sum));  
}
```

Реактивная обработка компактно записывается и понятно выглядит. Это большой плюс библиотеки. Но стоит рассмотреть ее тестируемость и понять, насколько легко тестировать определенную обработку. Начнем с наивного подхода, при котором логика тестирования определяется с нуля. Пример демонстрирует наличие проблем и подчеркивает необходимость в специализированной тестовой библиотеке.

Сконструируем поток из трех значений: 1, 2, 3, как показано в листинге 9.12. Затем делается 10-секундная пауза перед проверкой логики формирования окна. Обратите внимание: использовать метод `Thread.sleep()` в тестах не рекомендуется, но вскоре вы увидите, как исправить этот недочет. Наконец, мы проверяем, что полученное значение равно 6.

Листинг 9.12. Тестирование реактивной обработки: обобщенное решение

```
// Дано  
Flux<Integer> data = Flux.fromIterable(Arrays.asList(1, 2, 3));
```

```
Thread.sleep(10_000);

// Если
Flux<Integer> result = sumElementsWithinTimeWindow(data);

// To
assertThat(result.blockFirst()).isEqualTo(6);
```

К сожалению, в логике есть проблемы. Сначала в ней используется приостановка потока, которая увеличивает время на выполнение модульного теста. В реальной системе пришлось бы провести намного более масштабное тестирование и отработать больше сценариев. Это увеличило бы время, необходимое для всех модульных тестов, до неприемлемого уровня. Во-вторых, применение такого подхода к тестированию усложняет проверку в более сложных сценариях. Например, как убедиться, что значение после 10 секунд не будет учтено? Нужно выдать другое значение, подождать некоторое время, а потом проверить результаты. Анализируя этот простой пример, мы видим, что даже при использовании хорошей библиотеки без тестовой инфраструктуры провести тестирование крайне сложно, а иногда и вовсе невозможно.

К счастью, библиотека, используемая в этой главе, предоставляет тестовую библиотеку. Для реактивного тестирования используется библиотека `reactor-test` (<http://mng.bz/8lPz>). Это позволяет упростить тесты и делает возможным тестирование более сложных сценариев.

Для тестов мы используем класс `TestPublisher`, который позволяет предоставить данные для реактивного потока данных (листинг 9.13). Также с его помощью можно моделировать задержки без фактического замедления общего времени выполнения запросов. Приостановка для этого не нужна, поэтому тесты будут завершаться практически мгновенно. `TestPublisher` передается классу `StepVerifier`. Оба класса предоставляются библиотекой реактивного тестирования, которая совместима с реактивной рабочей библиотекой.

Листинг 9.13. Тестирование реактивной обработки с использованием тестовой библиотеки

```
final TestPublisher<Integer> testPublisher = TestPublisher.create();

Flux<Integer> result = sumElementsWithinTimeWindow(testPublisher.flux());

StepVerifier.create(result)
    .then(() -> testPublisher.emit(1, 2, 3))
    .thenAwait(Duration.ofSeconds(10))
    .then(() -> testPublisher.emit(4))
    .expectNext(6)
    .verifyComplete();
```

Класс `StepVerifier` позволяет сгенерировать значения, выждать заданное время без блокирования, а затем сгенерировать значения снова. В нашем тестовом

сценарии снова будут сгенерированы значения 1, 2, 3. После этого имитируется 10-секундная задержка, равная размеру окна. Затем генерируется еще одно значение. Наконец, мы проверяем, что первое произведенное значение равно 6. Таким образом, значение, сгенерированное после длины окна, не было включено в первое окно.

Этот подход позволяет протестировать любой сценарий, который приходится учитывать. Кроме того, сам факт тестирования задержки не означает, что модульное тестирование будет занимать больше времени. Тесты будут выполняться быстро, и мы сможем создать много модульных тестов для покрытия логики, реализованной с использованием реактивной библиотеки.

ПРИМЕЧАНИЕ

Многие библиотеки предоставляют в распоряжение разработчика тестовую библиотеку. Часто ее наличие становится признаком высокого качества и упрощает разработку.

Рассмотрим второй аспект тестируемости для сторонних библиотек — способы внедрения ложных объектов (*fake*) или имитаций (*mock*).

9.3.2. Тестирование с использованием объектов *fake* (тестовых двойников) и *mock*

Другой важный аспект, на котором следует сосредоточиться при принятии решения об использовании сторонней библиотеки, — возможность внедрения объектов, предоставленных пользователем, для целей тестирования. Таким объектом может быть имитация *mock*, с помощью которой можно смоделировать и проверить конкретное поведение, или объект *fake* (тестовый двойник), позволяющий предоставить данные или контекст тестируемому коду. Часто библиотеки скрывают слишком много внутренних подробностей от источника вызова, защищаясь от потенциальных злоупотреблений со стороны пользователей. Однако это может затруднить тестирование библиотеки.

Если вам доступна кодовая база библиотеки, найдите в ней точку создания нового экземпляра. Невозможность внедрения альтернативной реализации для целей тестирования указывает на будущие проблемы с тестированием. При использовании закрытой библиотеки, не раскрывающей свой исходный код, анализ может оказаться невозможным. В таких случаях эксперименты с тестами и проверка предположений становятся еще более важными, так как исходный код недоступен.

Займемся тестируемостью сторонней библиотеки независимо от того, обеспечивает она возможность внедрения тестового двойника от вызывающей стороны или нет. Допустим, вы хотите выбрать стороннюю библиотеку, которая предоставляет приложению функциональность кэширования. Один из самых

важных сценариев использования кэша — вытеснение старых элементов. Оно может основываться как на размере кэша, так и на продолжительности пребывания элемента в кэше, а также на совокупности этих условий. При оценке новой библиотеки стоит протестировать ожидаемое поведение, чтобы проверить свои предположения относительно него.

Начнем эксперимент с построения простого кэша, который получает ключ и преобразует его к верхнему регистру. В реальных системах используется более сложное поведение загрузчика кэша, но для наших целей достаточно тривиального примера, представленного ниже.

Мы хотим проверить поведение библиотеки на основании наших предположений. В следующем листинге строится новый кэш со сроком жизни после записи, равным `DEFAULT_EVICTION_TIME`. `CacheLoader` получает значение для ключа, предоставленного пользователем.

Листинг 9.14. Исходный вариант использования кэша

```
public class CacheComponent {
    public static final Duration DEFAULT_EVICTION_TIME = Duration.ofSeconds(5);
    public final LoadingCache<String, String> cache;

    public CacheComponent() {
        cache =
            CacheBuilder.newBuilder()
                .expireAfterWrite(DEFAULT_EVICTION_TIME)
                .recordStats()
                .build(
                    new CacheLoader<String, String>() {
                        @Override
                        public String load(@Nullable String key) throws Exception {
                            return key.toUpperCase();
                        }
                    }
                );
    }

    public String get(String key) throws ExecutionException {
        return cache.get(key);
    }
}
```

Логика выглядит прямолинейно, но все равно следует проверить предположения относительно ее поведения. Код библиотеки написан не нами, поэтому он может преподнести сюрпризы.

Требуется протестировать стратегию вытеснения используемого кэша. Для этого нужно смоделировать задержку между вставкой элемента кэша и проверкой процесса вытеснения. Поэтому мы должны ожидать столько, сколько идет вытеснение. В нашем сценарии использования это 5 секунд. В реальных системах

оно может длиться намного дольше (часы и даже дни). В следующем листинге показан исходный, наивный подход к тестированию, требующий использования `Thread.sleep()` с ожиданием `DEFAULT_EVICTION_TIME`.

Листинг 9.15. Тестирование без внедрения

```
// Дано
CacheComponent cacheComponent = new CacheComponent();

// Если
String value = cacheComponent.get("key");

// То
assertThat(value).isEqualTo("KEY");

// Если
Thread.sleep(CacheComponent.DEFAULT_EVICTION_TIME.toMillis());

// То
assertThat(cacheComponent.get("key")).isEqualTo("KEY");
assertThat(cacheComponent.cache.stats().evictionCount()).isEqualTo(1);
```

Обратите внимание: вытеснение производится при операции загрузки (`get`-методе). Чтобы инициировать его, необходимо вызвать метод доступа. Это один из неожиданных аспектов, который не соотносится с предположениями относительно библиотеки. Без качественного модульного теста это поведение, скорее всего, обнаружить не удастся. Как уже говорилось, если время вытеснения компонента слишком велико, тестирование компонента кэширования может стать неприемлемым. Нужно обдумать исходный код сторонней библиотеки (и возможно, просмотреть его), чтобы найти компонент, влияющий на поведение тестирования.

После беглого анализа выясняется, что `LoadingCache` при выполнении операции чтения использует объект `Ticker` для определения того, должно ли значение быть вытеснено. Доказательства приводятся в следующем листинге.

Листинг 9.16. Анализ тестируемости библиотеки кэширования

```
V get(K key, int hash, CacheLoader<? super K, V> loader) throws
    ExecutionException {
    ...
    long now = this.map.ticker.read();
    ...
}
```

Действительно, листинг показывает, что используемая сторонняя библиотека кэширования инкапсулирует логику времени в компоненте `Ticker`. Последнее, что нужно сделать для улучшения модульного теста для этой библиотеки, — проверить, может ли пользователь внедрить данный компонент. Это позволяет

предоставить суррогатную реализацию и повлиять на возвращаемые ей миллисекунды. Тем самым мы можем смоделировать течение времени без необходимости ожидания. К счастью, строитель `LoadingCache` содержит метод для передачи такого компонента извне, как показано в следующем листинге.

Листинг 9.17. Внедрение компонента, предоставленного пользователем

```
public CacheBuilder<K, V> ticker(Ticker ticker) {
    Preconditions.checkState(this.ticker == null);
    this.ticker = (Ticker)Preconditions.checkNotNull(ticker);
    return this;
}
```

Этот метод можно использовать в модульном тесте, передавая предоставленный пользователем объект `ticker` через строителя. Прежде всего необходимо реализовать интерфейс `Ticker`, который принимается строителем кэша. Благодаря тому что интерфейс хорошо спроектирован и прост, для него можно легко создать ложную реализацию. Если сторонний компонент позволяет внедрить собственную реализацию, но требует реализации интерфейса или расширения класса с большим количеством методов, добиться нужного поведения fake-объекта сложнее. Для этого требуются обширные знания о внутреннем компоненте, состоянии и моделируемых методах.

В листинге 9.18 `FakeTicker` использует `AtomicLong` для возвращения наносекунд. Важно использовать правильную единицу, определяемую контрактом сторонней библиотеки. Этот fake-объект позволит сдвинуть время на произвольную единицу в будущем или прошлом.

Листинг 9.18. Улучшение тестируемости за счет использования объекта fake, предоставленного пользователем

```
public class FakeTicker extends Ticker {
    private final AtomicLong nanos = new AtomicLong();

    public FakeTicker advance(long nanoseconds) {
        nanos.addAndGet(nanoseconds);
        return this;
    }

    public FakeTicker advance(Duration duration) {
        return advance(duration.toNanos());
    }

    @Override
    public long read() {
        return nanos.get();
    }
}
```

Поскольку в тестах можно использовать `FakeTicker`, необходимость в `Thread.sleep()` отпадает, так что модульные тесты могут быть быстрыми и покрывать множество сценариев использования. Этот новый механизм может использоваться (как показано в следующем листинге) для проверки более широкого спектра предположений об этой библиотеке.

Листинг 9.19. Улучшение тестирования с использованием fake-объектов

```
// Дано
FakeTicker fakeTicker = new FakeTicker();
CacheComponent cacheComponent = new CacheComponent(fakeTicker);

// Если
String value = cacheComponent.get("key");

// To
assertThat(value).isEqualTo("KEY");

// Если
fakeTicker.advance(CacheComponent.DEFAULT_EVICTION_TIME);

// To
assertThat(cacheComponent.get("key")).isEqualTo("KEY");
assertThat(cacheComponent.cache.stats().evictionCount()).isEqualTo(1);
```

Тест стал значительно лучше. Появилась возможность моделировать ход времени при помощи метода `advance()`. Даже если время вытеснения составляет несколько дней, модульный тест завершится мгновенно.

Представьте сценарий, в котором тестируемая сторонняя библиотека не дает возможности внедрить компонент `Ticker`, используемый во внутренней реализации. В таком случае невозможно проверить некоторые из предположений. Если вы решите задействовать эту библиотеку, возникнут проблемы, потому что не получится протестировать некоторые аспекты поведения. По этой причине, скорее всего, выбор падет на другую библиотеку.

Почти каждая сторонняя библиотека имеет некоторое внутреннее состояние. Если библиотека позволяет внедрить другую реализацию, это дает ей значительное преимущество в отношении тестируемости.

ПРИМЕЧАНИЕ

Если тестируемая сторонняя библиотека имеет зависимости, которые трудно тестировать, попробуйте использовать `Mockito`, `Spock` или другие тестовые фреймворки. Они могут упростить тестирование некоторых граничных случаев.

Пока что мы обсуждали модульное тестирование сторонних библиотек. Теперь рассмотрим возможности выполнения интеграционного тестирования стороннего кода. Это также может повлиять на решение о выборе той или иной библиотеки.

9.3.3. Набор инструментов интеграционного тестирования

После того как мы убедились, что интересующая нас сторонняя библиотека обеспечивает возможность модульного тестирования, можно перейти на следующий уровень: к интеграционным тестам. Предположим, что импортируемая библиотека предоставляет функциональность, которую можно изолировать от других компонентов. В этом случае вполне достаточно провести модульное тестирование и положиться на интеграционные тесты, которым не обязательно знать о фактической реализации. В основе интеграционных тестов лежит принцип тестирования высокоуровневых компонентов без учета низкоуровневых подробностей. Однако приложения обычно строятся на базе фреймворков, предоставляющих разнообразную функциональность. В JVM можно использовать Spring, Dropwizard, Quarkus, OSGi или Akka (список далеко не полон). Такие фреймворки могут иметь несколько зависимых компонентов, предоставляющих уровень API (HTTP), уровень доступа к данным, фреймворк внедрения зависимостей и т. д.

Также стоит заметить, что у этих компонентов может быть собственный жизненный цикл. Начать работу над приложением с заданным фреймворком относительно несложно, но при этом необходимо создать правильные компоненты и внедрить их. Более того, иногда конфигурация приложения для интеграционных тестов отличается от конфигурации нормального запуска. Это может быть другая строка подключения к базе данных, другие имена пользователей, другие пароли и т. д.

Когда вы начинаете строить приложение на базе какого-то фреймворка, следует убедиться в том, что вы можете легко управлять приложением в интеграционных тестах. Например, фреймворк Spring позволяет запустить приложение в интеграционных тестах с использованием аннотации `@SpringBootTest` (<http://mng.bz/ExPd>) и `SpringRunner` (<http://mng.bz/NxPn>), как показано в следующем листинге.

Листинг 9.20. Интеграционные тесты Spring

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles("integration")
public class PaymentServiceIntegrationTest {

    @Value("${local.server.port}")
    private int port;

    private String createTestUrl() {
        return "http://localhost:" + port + suffix;
    }
    // ...
}
```

← Внедряет переданный порт

Фреймворк Spring предоставляет несколько способов выполнения тестов. В листинге 9.20 использовалось тестирование из библиотеки Spring Boot со всеми необходимыми аннотациями. Если приложение основано на этом фреймворке, `@SpringBootTest` находит все компоненты и запускает их с соответствующим жизненным циклом. Не нужно беспокоиться о фактической процедуре запуска. Более того, при необходимости в тестировании HTTP API будет запущен встроенный веб-сервер HTTP со свободным портом. Порт внедряется после запуска сервера. (Думать о выборе свободного порта не нужно — тестовая библиотека Spring позаботится об этом.) Наконец, можно выполнить нормальные запросы HTTP к конечной точке `localhost`, созданной методом `createTestUrl()`.

Также заметим, что можно активировать разные профили в интеграционных тестах. Это полезно, если вам нужны разные конфигурации компонентов, инициализированные для интеграционных тестов. В тестовой библиотеке Spring возможность выбора разных профилей при выполнении тестов встроена и предоставляется автоматически.

Кажется, что запуск встроенного сервера HTTP и предоставление конечных точек HTTP не создает особых проблем. Однако реальные приложения обычно устроены сложнее. В них существуют уровни доступа к данным с репозиториями, интеграция с другими сервисами и множество других компонентов. Если фреймворк предоставляет библиотеку интеграционного тестирования, экспериментировать с библиотекой и анализировать ее будет быстрее и проще. В следующем разделе мы сосредоточимся на проблеме избытка зависимостей в сторонних библиотеках, которые могут серьезно повлиять на работу приложения.

9.4. ЗАВИСИМОСТИ СТОРОННИХ БИБЛИОТЕК

Все импортируемые и используемые в коде библиотеки или фреймворки написаны разработчиками, принимавшими то же решение: стоит ли реализовать небольшую часть логики самостоятельно или же воспользоваться другой библиотекой, предоставляющей этот функционал? Очевидно, что импортируемая библиотека, которая обеспечивает, скажем, функциональность клиента HTTP, не должна зависеть от другой библиотеки, предлагающей то же самое. Ситуация немного меняется, если создатели библиотеки не проработали ее основную функциональность. Например, клиентская библиотека HTTP может предоставлять встроенные средства сериализации и десериализации JSON, как показано на рис. 9.6.

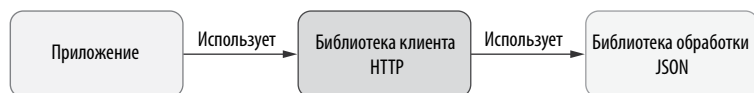


Рис. 9.6. Клиент HTTP с обработкой JSON

Обработка JSON — нетривиальная область, и разработчики библиотеки клиента HTTP могут не быть в ней экспертами. Возможно, поэтому они и решили использовать другую стороннюю библиотеку, которая предоставит этот функционал. Решение логичное, но оно создает проблемы в нашем приложении.

Все усложняется, если приложению требуется использовать обработку JSON в логике, не связанной с библиотекой клиента HTTP. Нужно помнить, что каждый класс, поставляемый с клиентом HTTP (включая их зависимости), будет виден коду приложения. Это позволяет использовать библиотеку обработки JSON через транзитивную зависимость, которую применяет библиотека клиента HTTP. Тем не менее делать этого не стоит по нескольким причинам.

Дело в том, что код приложения сильно связывается с библиотекой, используемой сторонней библиотекой. В будущем библиотека клиента HTTP может сменить библиотеку обработки JSON. В этом случае в коде возникнут проблемы, потому что библиотека уже не будет предоставлять исходную библиотеку JSON и ее классы.

9.4.1. Предотвращение конфликтов версий

Другое (более правильное) решение — создать прямую зависимость от приложения к библиотеке обработки JSON, которую планируется использовать (рис. 9.7). К сожалению, в этом случае тоже не избежать проблем, потому что между двумя библиотеками JSON может возникнуть конфликт версий. Это произойдет, если клиент HTTP и приложение используют разные версии библиотеки JSON.

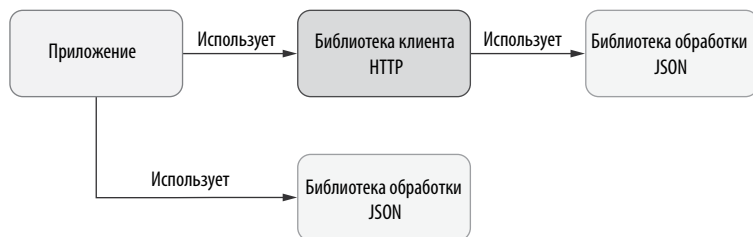


Рис. 9.7. Приложение использует библиотеки HTTP и JSON напрямую

В нашем примере важно заметить, что реальные приложения часто имеют много сторонних зависимостей. Каждая из них может добавить собственные зависимости, и тогда ситуация быстро выйдет из-под контроля. В одном приложении могут использоваться разные версии нескольких библиотек.

Все классы, предлагаемые библиотекой JSON, предоставляются в пакете `com.fasterxml.jackson`. Например, при желании в приложении можно использовать `com.fasterxml.jackson.databind.ObjectMapper`. Это означает, что обе версии

этого класса — из клиента HTTP и из приложения — доступны в этом пакете. При этом программа сборки должна выбрать один из этих классов. Как будет показано в главе 12, из-за этого могут возникать разные проблемы: метод не найден, сигнатура метода изменилась и т. д.

СЕМАНТИЧЕСКОЕ ВЕРСИОНИРОВАНИЕ И СОВМЕСТИМОСТЬ

В большинстве библиотек принято семантическое версионирование со строкой версии, состоящей из трех частей: основной, дополнительной и обновления. Любые критические изменения, способные нарушить работоспособность, должны обозначаться изменением основной части строки версии. Эта тема намного подробнее рассматривается в главе 12, но сейчас важно отметить, что, если полный набор зависимостей использует одну основную версию библиотеки JSON, можно использовать только последнюю из этих версий. Если в приложении задействовано несколько основных версий, они должны быть разными зависимостями.

К счастью, эта проблема решается в импортируемой сторонней библиотеке. Этот прием называется *замещением* (shading). Объясним его на примере с клиентом библиотеки HTTP. Рассмотрим ситуацию, в которой используется библиотека FasterXml Jackson (<https://github.com/FasterXML/jackson>).

Если клиент HTTP использует метод замещения для своей зависимости JSON, он может переписать все имена пакетов и разместить их под другим префиксом. Допустим, клиент HTTP предоставляет доступ ко всем своим классам в иерархии `com.http.client`. В этом случае после замещения все классы библиотеки JSON из библиотеки клиента HTTP будут доступны по имени пакета `com.http.client.com.fasterxml.jackson`.

Этот прием позволяет клиенту HTTP *скрыть* классы библиотеки обработки JSON от приложения. Они все еще доступны, но в приложении можно использовать независимую версию Jackson. Думать о зависимостях, внесенных сторонней библиотекой клиента HTTP, не придется.

Замещение — полезный метод, но он требует от разработчиков сторонних библиотек существенных затрат на обслуживание. Процесс замещения, переписывающий классы, должен быть применен для каждой сторонней библиотеки, которую необходимо скрыть. Это происходит в фазе сборки библиотеки. Как следствие, процесс сборки усложняется, потому что он может потребовать определения поведения замещения для нескольких библиотек. Если замещаемая сторонняя библиотека изменит свою схему пакетов, то конфигурацию замещения необходимо будет адаптировать.

При оценке библиотеки для использования в проекте следует изучить все ее зависимости. Если в ней используется замещение, это значит, что библиотека скрывает сторонние зависимости от приложения. Следовательно, приложение не будет загрязнено лишними зависимостями. Это значительное преимущество

перед другими библиотеками, которые предоставляют ту же функциональность, но не скрывают сторонние зависимости (с которыми замещение не используется). Также замещение — это признак того, что используемая сторонняя библиотека хорошо спроектирована и тщательно продумана.

9.4.2. Слишком много зависимостей

Нужно отдавать себе отчет в том, что почти каждая библиотека должна использовать другие библиотеки для предоставления вспомогательной функциональности. Поэтому следует проверить количество библиотек, которые она с собой приносит. Существует большая разница между импортированием других библиотек для сложной функциональности, которую трудно написать самостоятельно, и для простых задач, которые легко решить и реализовать с нуля.

Следует помнить, что каждая импортируемая библиотека влияет на приложение. Часто создатели библиотек при всем желании не могут выполнить замещение для всех зависимостей, так как это потребует слишком много времени и усилий.

Каждая зависимость, импортируемая в приложение, оказывает на него влияние. В самом частом варианте развертывания приложений создается автономный пакет, который содержит все необходимые зависимости (fat jar, или uber jar) (рис. 9.8).



Рис. 9.8. Структура автономного пакета fat jar

Если предположить, что код приложения занимает 20 Мбайт, все равно придется упаковать исполнительную среду Java и все используемые сторонние библиотеки. В результате будет создано автономно запускаемое приложение. В нашем примере оно займет 120 Мбайт.

Когда приложение развертывается в формате fat jar, оно всегда выполняется напрямую, без внешних зависимостей. Этот способ приобрел особую популярность в контейнерных средах (например, Kubernetes и Docker).

Это значит, что создаваемый файл с приложением будет увеличиваться с каждой содержащейся в нем зависимостью, включая все сторонние зависимости и зависимости, используемые сторонними библиотеками. По этой причине следует обращать особое внимание на количество зависимостей приложения. Чем меньше зависимостей, тем меньше приложение. Чем меньше приложение, тем быстрее оно запускается, тем проще его развертывать и им управлять.

Время выполнения также уменьшается, потому что построенное приложение должно быть загружено в память машины, на которой оно выполняется. Этот аспект привлекает все больше внимания, так как бессерверные решения набирают популярность. В таких решениях среда выполнения приложений ограничена (например, процессор и память). Кроме того, в бессерверных средах очень важно время запуска.

В экосистеме Java плагин Maven-shade-plugin (<https://maven.apache.org/plugins/maven-shade-plugin/>) упрощает процесс построения пакетов fat jar. Также он позволяет выполнять замещение с использованием переименования. В следующем разделе речь пойдет о том, как выбрать сторонний код для приложения, а также о вариантах его использования.

9.5. ВЫБОР И ОБСЛУЖИВАНИЕ СТОРОННИХ ЗАВИСИМОСТЕЙ

Выбор библиотеки, которая будет использоваться в коде, всегда создает некоторый уровень связанности между кодом приложения и сторонней библиотекой. Стороннюю библиотеку можно скрыть за уровнем абстракции и предоставить методы, вызываемые из кода, но это повышает затраты на обслуживание. Да, этот способ работает. Библиотеки, которые мы используем, имеют собственный жизненный цикл разработки.

9.5.1. Первые впечатления

Когда вы только рассматриваете возможность использования библиотеки и включения ее в свое приложение, довольно легко проверить ряд аспектов, прежде чем погружаться в сложности определения технической пригодности. Как правило, это проще сделать в отношении библиотек с открытым кодом. Однако на многие из перечисленных ниже вопросов можно ответить и для коммерческих библиотек. Конечно, не стоит забывать и о серьезном анализе аспектов, речь о которых шла в предыдущих разделах.

- *Насколько библиотека стабильна?* Если у нее еще нет стабильной версии, уверены ли вы, что она появится, когда вам понадобится поставлять код?

- *Находится ли библиотека в активной разработке?* Если библиотека решает ограниченный набор задач, вполне нормально, что она фактически готова и не требует обновлений в течение какого-то времени. И все же стоит убедиться в том, что она не заброшена.
- *Популярна ли библиотека в сообществе?* Если у библиотеки активная экосистема, вам будет намного проще получить помощь, к тому же это, как правило, верный признак качества кода.
- *Кто занимался созданием библиотеки?* Библиотеку, созданную коллективом разработчиков (возможно, при поддержке крупной компании, которая сама ее использует), выбирать менее рискованно, чем проект «для души», написанный одиночкой.
- *Хорошо ли документирована библиотека?* Проверьте наличие справочной документации API, концептуальной документации, учебников или руководства по быстрому началу работы.

И хотя все эти вопросы вполне показательны, ответы на них не стоит рассматривать как категоричные «да» или «нет». Большие компании отказываются от библиотек, которые они разрабатывали годами, одиночки десятилетиями ответственно ведут проекты, а иногда можно успешно пользоваться библиотекой при скудной документации. Это всего лишь вопросы, которые необходимо учитывать при принятии решения. Рассмотрим некоторые аспекты более подробно, начиная с того, как адаптировать сторонний код к имеющейся базе данных.

9.5.2. Разные подходы к повторному использованию кода

Как вы успели убедиться, выбор библиотеки — непростая задача. Необходимо учесть множество факторов: конфигурацию, параллелизм, модели масштабирования, тестируемость, количество зависимостей, жизненный цикл разработки, версионирование и т. д. Если задача, которую должна решать библиотека, достаточно сложна и вы планируете использовать разные ее возможности, выбор сторонней библиотеки оправдывает все компромиссы и затраты на сопровождение. Но если вам нужна отдельная мелкая функция — допустим, форматирование строки или расширение набора вспомогательных методов коллекций, — стоит подумать о возможности реализации конкретного метода в кодовой базе. С другой стороны, выбранный метод сторонней библиотеки может немного упростить код. Однако с внедрением метода в кодовую базу добавляются и проблемы с обслуживанием и сложностью.

В зависимости от лицензии выбранной сторонней библиотеки можно скопировать нужный метод в кодовую базу, добавить для него модульные тесты и стать полноправным владельцем кода. Это позволит использовать готовый код, проверенный реальной эксплуатацией, без необходимости импортировать библиотеку

со всеми ее API и методами. Такое решение может быть рациональным, если нужно использовать небольшие части сторонней библиотеки. Конечно, у такого решения есть и обратная сторона — вы несете ответственность за исправление ошибок. Но если код, который вы портируете, относительно мал и вы в полной мере понимаете его, особых проблем быть не должно.

Можно также клонировать исходную библиотеку и разработать необходимую функциональность, если изменить исходный код трудно или невозможно. При этом возникает множество проблем с обслуживанием. Например, через какое-то время две версии кода могут разойтись, после чего согласовать их станет затруднительно или даже невозможно.

9.5.3. Привязка к производителю

Важно учитывать, что независимо от популярности конкретной библиотеки и стоящей за ней компании, используемое решение может быть заброшено и в конечном итоге рискует исчезнуть. Представьте, что облачный сервис заменили улучшенной версией или запатентованным программным продуктом, но создавшая его компания была продана, и продукт изменился. Или другой вариант: сторонняя библиотека с открытым кодом пользовалась популярностью, и вы решили задействовать ее в своем приложении. Но через какое-то время появляется новая библиотека, которая решает ту же задачу быстрее и лучше, и люди начинают переходить на новое решение. Прежнее решение перестает развиваться и затем переходит в режим сопровождения — его разработка прекращается.

Архитектуры развиваются, создаются новые паттерны, программные продукты устаревают. Когда вы начинаете пользоваться новой библиотекой или сервисом, следует учитывать, что в будущем может возникнуть необходимость в переходе на новое решение. Вероятность этого не одинакова для всех программных компонентов, которые вы используете. Если вы знаете, что эта вероятность велика, точки интеграции с этими библиотеками (или сервисами) следует скрыть за прослойкой абстракции. Когда возникнет необходимость в переключении реализации, изменение не распространится на множество точек кода, а будет инкапсулировано в абстракции, и изменения в коде нужно будет вносить только в ней.

Начиная работать с новой библиотекой, необходимо следить за тем, как она влияет на приложение и архитектуру. Чем глубже интеграция, тем сложнее будет сменить производителя в будущем. В реальном приложении сложно (или невозможно) скрыть каждую библиотеку и точку интеграции с сервисом за абстракцией. Определенный уровень привязки к производителю всегда будет трудно преодолеть, но можно постараться свести ее к минимуму, выбирая библиотеки, не требующие сильной связанности с приложениями.

9.5.4. Лицензирование

Принимая решение об использовании кода другой библиотеки, необходимо учитывать условия ее лицензии. Допустим, вы выбираете библиотеку с лицензией GNU General Public License. Чтобы использовать ее код в проекте, вам, возможно, придется опубликовать исходный код своего проекта. Это может стать преградой для многих внутренних проектов, код которых вы не хотите раскрывать. Решения по лицензированию сложны, и ошибки обходятся дорого. В случае сомнений я рекомендую проконсультироваться с юристами, которые дадут необходимые рекомендации.

9.5.5. Библиотеки и фреймворки

Часто в начале работы с библиотекой в кодовой базе ее можно абстрагировать без существенных затрат. Например, все вызовы сервиса HTTP с использованием конкретной библиотеки могут быть скрыты в специальном классе сервиса. Затем все взаимодействия между библиотекой HTTP и кодом приложения могут осуществляться через класс сервиса, а не напрямую через библиотеку HTTP. Тем не менее необходимо действовать внимательно и не раскрывать никаких конкретных подробностей, относящихся к используемой библиотеке, в том числе исключения (как было показано в главе 3) и конфигурацию (главу 6). Если это сделать, вам будет проще изменить решение позднее. Вы сможете переключиться на библиотеку с другой реализацией без существенных затрат. Также можно выбрать вариант с самостоятельной реализацией и устранением зависимости от этой библиотеки.

Ситуация сильно меняется при использовании фреймворков. Как правило, они оказывают значительное влияние на код приложений. Некоторые фреймворки агрессивны, они требуют обязательного использования своих конструкций в наших приложениях. В этом можно легко убедиться, просматривая список импортирования в кодовой базе. Чем больше команд импортирования фреймворка в кодовой базе, тем сильнее его привязка к нашему приложению. Фреймворк намного сложнее заменить в течение жизненного цикла приложения, чем библиотеку. Поэтому нужно быть аккуратнее и провести более тщательный анализ перед выбором фреймворка для приложения (по сравнению с выбором библиотеки).

9.5.6. Безопасность и обновления

Последний (но не менее важный) аспект, о котором следует упомянуть, — последствия использования сторонних библиотек для безопасности приложений. Как известно, любой программный продукт может содержать баги. Они влияют не только на правильность работы и производительность, но и на общую

безопасность приложений. По этой причине необходимо выполнять тесты безопасности перед развертыванием новой версии продукта. Средства автоматического сканирования безопасности кодовой базы помогают находить проблемы, но не стоит забывать, что используемые библиотеки становятся кодом.

Каждая используемая сторонняя зависимость развивается и может содержать свои дефекты безопасности. Когда в сторонней библиотеке обнаруживается новая уязвимость, ее авторы должны разобраться с ней как можно скорее. Чаще всего это приводит к выпуску новой версии вскоре после обнаружения уязвимости. Когда проблема решена, проблемную библиотеку в кодовой базе следует обновить как можно быстрее. Чем дольше мы ждем, тем больше времени будет у потенциального злоумышленника для использования уязвимости.

Как узнать, что в сторонней зависимости обнаружена проблема безопасности? Например, можно поискать данные об используемой библиотеке на веб-сайтах с информацией об уязвимостях безопасности (скажем, <https://www.cvedetails.com/>). Там публикуются обновления, связанные с безопасностью различных продуктов и библиотек. Впрочем, ручная проверка сайтов — занятие утомительное и долгое. К счастью, можно автоматизировать проверки безопасности, которые сканируют все сторонние библиотеки и оповещают об обнаруженных проблемах. Некоторые средства такого рода (например, <https://dependabot.com/>) даже могут автоматически обновлять версию библиотеки-нарушителя и инициировать изменения (например, pull-запросы с использованием Git) в кодовой базе.

Обновления безопасности — самое важное, но стоит следить и за другими обновлениями библиотеки. При выходе новой полноценной версии можно выяснить, сколько всего изменилось, и запланировать обновление в ближайшем будущем, особенно если более старая версия имеет ограниченный срок поддержки.

К дополнительным версиям проще адаптироваться, если библиотека правильно использует семантическое версионирование; может оказаться, что новые возможности позволяют упростить часть кода. Также стоит проверить, какие баги были исправлены в новых выпусках. Бывает, что в приложении существует проблема, о которой вы даже не подозреваете.

9.5.7. Список решений

Если вы хотите задействовать больше функций сторонней библиотеки (или решение с копированием кода невозможно), воспользуйтесь контрольным списком того, что нужно проверить. Это позволит избавиться от многих проблем в будущем.

- *Возможности настройки конфигурации и значения по умолчанию* — можно ли передать (и переопределить) все важнейшие настройки?

- *Модель параллельного выполнения, масштабируемость и производительность* — предоставляет ли библиотека асинхронный API, если приложения также используют асинхронную модель выполнения?
- *Распределенный контекст* — будет ли приложение безопасно работать в распределенном контексте (на нескольких узлах)?
- *Анализ надежности* — выбираете ли вы фреймворк или библиотеку? Если это фреймворк, требуются более основательные исследования.
- *Проведение модульных и интеграционных тестов для проверки предположений относительно библиотеки* — насколько сложно тестировать код, использующий библиотеку? Предоставляет ли она собственный инструментарий тестирования?
- *Зависимости* — от чего зависит библиотека? Существует ли она автономно и изолированно? Или она загружает множество внешних зависимостей, влияя на размер и сложность приложения?
- *Версионирование* — использует ли библиотека семантическое версионирование? Развивается ли она с сохранением обратной совместимости?
- *Сопровождение* — является ли библиотека популярной и активно поддерживаемой?
- *Интеграция* — насколько сильно библиотека влияет на кодовую базу при интеграции? Насколько велик риск привязки к одному производителю?
- *Лицензирование* — допускает ли лицензия сторонней библиотеки применение в вашем контексте?
- *Безопасность и обновления* — часто ли выходят обновления используемых компонентов для решения проблем с безопасностью?

ИТОГИ

- Большинство библиотек, которые мы используем, нуждаются в конфигурировании. Помните о значениях по умолчанию, которые могут повлиять на поведение кода.
- Предпочтительное использование соглашений вместо конфигурации упрощает прототипизацию и фазу разработки, но может скрывать проблемы, которые проявятся в условиях реальной эксплуатации.
- Приложение должно использовать сторонние библиотеки, предоставляющие схожую модель параллельного выполнения. Это позволяет создавать приложения с более высокой производительностью.
- В Java проще использовать асинхронный API в синхронном контексте, чем упаковывать асинхронный API в синхронную абстракцию. Создание асин-

хронной обертки для синхронного API добавляет значительную сложность в приложение.

- Выбор библиотеки с синхронной моделью может ограничить масштабируемость в будущем, если вы решите сменить модель выполнения.
- Масштабируемость сторонних библиотек может существенно меняться между сценарием с одним узлом и распределенной системой с N узлами. Проверяйте свои предположения относительно моделей масштабируемости библиотеки, которую вы планируете использовать, пока изменения не начнут обходиться слишком дорого.
- Используйте тестирование для проверки предположений относительно кода, который вам не принадлежит.
- Тестируемость играет исключительно важную роль при выборе сторонней библиотеки. Она также отражает общее качество используемого кода.
- Средства модульного и интеграционного тестирования, поставляемые с библиотеками, существенно упрощают и ускоряют тестирование кода.
- Все сторонние зависимости приносят свои зависимости. Помните об этом и анализируйте их, прежде чем импортировать код в приложение.
- Размер приложения становится более важным в контейнерных и бессерверных средах. Чем меньше приложение, тем быстрее оно развертывается.
- Сторонние библиотеки необходимо обновлять, чтобы своевременно интегрировать исправления багов, дефектов безопасности и производительности этих библиотек. Все используемые библиотеки должны реализовать механизм семантического версионирования; это упростит процесс обновления.
- Семантическое версионирование дает обширную информацию о рабочем цикле и ходе разработки сторонних библиотек. Когда меняется основная версия, вы знаете, что обновление не будет простым. Но при изменении дополнительной версии или версии патча в процессе обновления ничего сложного нет.

10

Целостность и атомарность в распределенных системах

В этой главе

- ✓ Трафик между микросервисами, развернутыми на N узлах, и распределенной базой данных.
- ✓ Приложения, корректно работающие в сценарии с одним узлом, и их доработка для правильной работы на N узлах.
- ✓ Различия между атомарностью и целостностью в среде приложения.

Если вы хотите, чтобы ваше приложение масштабировалось и выполнялось в распределенной среде, необходимо проектировать код соответственно. Целостность системы играет важную роль, и ее относительно легко достичь, если приложение развертывается на одном узле и использует стандартную базу данных. В таком контексте транзакции базы данных гарантируют атомарность операций. Однако на практике приложения должны быть масштабируемыми и эластичными.

В зависимости от паттернов трафика может возникнуть необходимость в развертывании приложения на N узлах. После того как приложение будет развернуто на N узлах, могут проявиться проблемы с масштабируемостью на нижнем уровне — базы данных. В таком случае часто приходится переводить уровень данных в распределенную базу данных. Это позволяет распределить обработку входного трафика по N микросервисам, что, в свою очередь, приводит к распределению трафика по M узлам базы данных. Для такой среды код следует проектировать совершенно иначе. В этой главе основное внимание уделяется

решениям и изменениям, необходимым для обеспечения целостности и атомарности логики приложения в распределенной среде.

Начнем с простой архитектуры с несколькими сервисами, в которой каждый сервис развертывается только на одном узле. Мы проанализируем и определим характеристики трафика в таком контексте. После этого мы постепенно перейдем к более сложным архитектурам и посмотрим, как при этом будут эволюционировать наши предположения о структуре системы.

10.1. ИСТОЧНИКИ ДАННЫХ С ДОСТАВКОЙ «НЕ МЕНЕЕ ОДНОГО РАЗА»

Не поддавайтесь соблазну внедрить упрощенное представление приложения, развернутого на одном узле, с использованием нераспределенной, стандартной базы данных SQL. Однако важно понимать, что даже если сервис задействует простейшую модель развертывания и не проектировался с расчетом на масштабирование, он может (и, вероятно, будет) работать в распределенной среде. Дело в том, что если система предоставляет бизнес-функциональность, ей почти наверняка придется вызывать другие сервисы. При каждом обращении к внешнему сервису выполняется сетевой вызов. Это означает, что сервис должен выполнить запрос по сети и дождаться ответа.

10.1.1. Трафик между сервисами с одним узлом

Допустим, приложение А, развернутое на одном узле, должно обратиться с вызовом к почтовому сервису. При получении запроса сервис отправляет сообщение электронной почты конечному пользователю. В таком случае приложение работает в распределенной среде.

Важно помнить, что любой сетевой вызов может завершиться неудачей (рис. 10.1). Отказ может быть вызван ошибкой вызываемого сервиса.

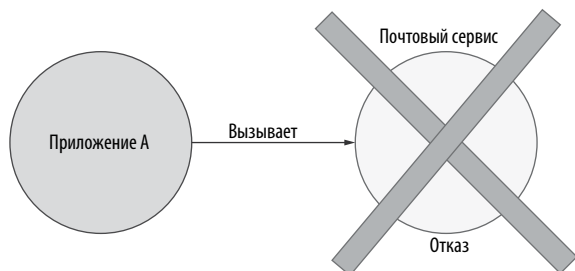


Рис. 10.1. Запрос к почтовому сервису завершается неудачей из-за отказа сервиса

На первый взгляд ситуация проста. Тем не менее ее довольно сложно анализировать с точки зрения вызывающей стороны (приложение А). В реальности приложение А может быть сервисом интернет-магазина, сервисом автоматизации маркетинга, сервисом подтверждения платежей и т. д. Сбои в почтовом сервисе происходят как до, так и после отправки сообщения. Если при отправке возникает сбой, а почтовый сервис дает обоснованный ответ (например, код статуса, сообщающий о перерыве на техническое обслуживание), приложение А может заключить, что сообщение не было отправлено. Но если приложение получает обобщенную ошибку без пояснений, можно предположить, что сообщение еще не доставлено.

Все становится еще сложнее, если принять во внимание вероятность сетевых сбоев. Возможна ситуация, в которой вы вызываете почтовый сервис, что приводит к успешной отправке сообщения. Почтовый сервис отвечает приложению А статусом, который сообщает, что все прошло успешно. Однако следует помнить, что запрос и ответ передаются по сети. Как уже говорилось, каждый сетевой вызов может завершиться неудачей по множеству разных причин. Например, может сломаться маршрутизатор, коммутатор или концентратор, находящийся на сетевом пути. Также возможно нарушение связности сети, которое препятствует доставке пакета (или ответа), как показано на рис. 10.2.

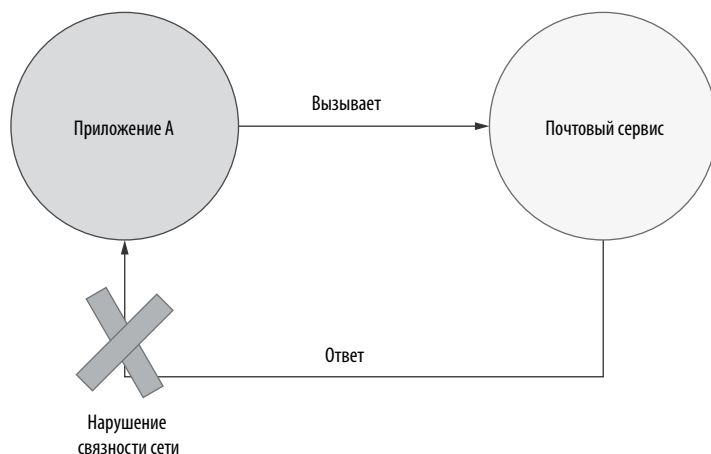


Рис. 10.2. Сетевой сбой при отправке ответа приложению А

При сетевом сбое вызывающая сторона приложения А не может предсказать исход операции. Приложение А заметит тайм-аут, показывающий, что ответ не был отправлен в течение некоторого ограниченного промежутка. По этой причине нарушается целостность состояния вызывающей стороны: у нее нет полного представления о системе. Возможно, сообщение было отправлено... а может и нет.

10.1.2. Повтор попытки вызова

Если приложение А не получило успешного ответа, оно может попытаться повторить исходный запрос — это одно из возможных решений. Если причиной сбоя было временное нарушение связности сети, с высокой вероятностью повторная попытка завершится неудачей. В этом случае вызывающее приложение снова получает (по большей части) целостное представление о системе.

Однако в архитектуре системы повторные попытки проблематичны. Представьте, что их будет несколько. Нельзя исключать, что почтовый сервис отправит ряд одинаковых сообщений, как показано на рис. 10.3. Рассмотрим ситуацию, в которой первый запрос завершился сбоем, затем первая повторная попытка тоже оказалась неудачной и запрос повторяется еще раз. Получается, что сообщение будет отправлено до трех раз! Причина в том, что вы не знаете, когда произошел отказ предыдущего вызова — до или после отправки сообщения.

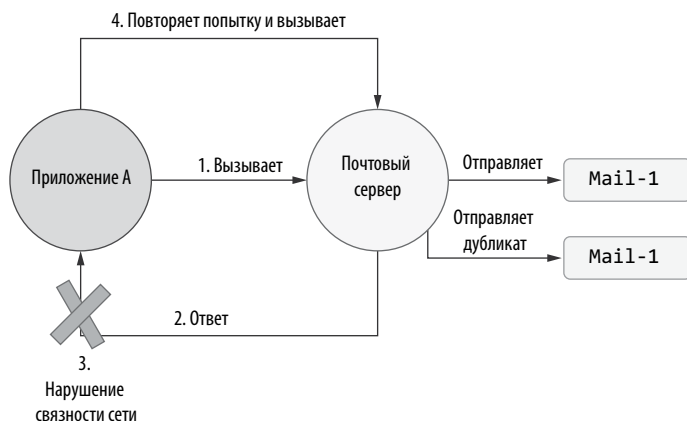


Рис. 10.3. Повторная попытка отправки запроса из приложения А приводит к появлению дубликатов сообщения

Рассмотрим последовательность действий на рис. 10.3. На первом этапе обработки приложение А отправляет запрос почтовому сервису, вызываемому по некоторому протоколу. Почтовый сервис успешно отправляет сообщение и после отправки возвращает ответ вызывающей стороне (приложение А). Однако во время передачи ответа нарушается связность сети. Приложение А воспринимает это как сбой. Вызывающее приложение не получает ответа, и с истечением тайм-аута происходит сбой. С точки зрения почтового сервиса повторная попытка — всего лишь очередной запрос, который необходимо обработать. Поэтому то же сообщение отправляется еще раз. На этот раз ответ успешно доставляется вызывающему приложению А, и повторной попытки не будет. К сожалению, сообщение было отправлено дважды.

В архитектурах реальных систем возникает необходимость в интеграции с несколькими внешними сервисами. Здесь повторные попытки отправки писем, скорее всего, не создадут проблем, хотя и могут привести к тому, что почта на компьютере клиента попадет в папку «Спам». Но возможны и более значительные проблемы, если, например, система должна выполнить оплату. Платеж также является внешним вызовом, а повторные попытки создают проблемы, потому что деньги могут быть списаны со счета пользователя два раза (или больше).

При повторной попытке операции почтового сервиса обеспечивается семантика доставки «не менее одного раза». Если приложение А повторяет операцию, пока она не завершится успехом, сообщение будет доставлено один или более раз. Это может привести к дублированию доставки, но ее отсутствие при этом исключается (не считая критического отказа всего почтового сервиса). Разные семантики доставки подробно обсуждаются в следующей главе. Пока достаточно знать, что наша архитектура работает по принципу доставки «не менее одного раза», что приводит к дублированию на стороне почтового сервиса при выполнении повторных попыток приложением.

10.1.3. Производство данных и идемпотентность

Повторные попытки выполнения операций, имеющих побочные эффекты, обычно небезопасны. Именно так обстоит дело в нашей текущей архитектуре. Но как определить, безопасна ли повторная попытка? На этот вопрос отвечает такая характеристика системы, как *идемпотентность*. Операция идемпотентна, если она приводит к одному и тому же результату независимо от количества вызовов.

Например, чтение информации из базы данных является идемпотентным (предполагается, что читаемые данные не изменяются между попытками). Все операции получения данных — например, от конечной точки HTTP — также должны быть идемпотентными. Если нашему сервису нужны данные от другого сервиса, он может повторить эту операцию несколько раз. Такой подход предполагает, что ни одна из операций получения данных не изменяет состояние. Получение значения безопасно, а повторять операцию можно сколько угодно.

Другой пример: удаление записи для заданного идентификатора также идемпотентно. Результат будет одинаков независимо от количества его выполнений. Если удалить запись с конкретным идентификатором, а потом снова попытаться это сделать, ничего не произойдет. Независимо от количества вызовов результат останется неизменным.

С другой стороны, производство данных чаще всего является неидемпотентной операцией. Отправка почты не является идемпотентной. Когда вы инициируете операцию отправки, письмо отсылается. В архитектуре, представленной

в предыдущем разделе, существует побочный эффект, который невозможно отменить. Повторная попытка выполнения такой операции приводит еще к одной отправке, и это еще один побочный эффект.

Стоит заметить, что некоторые действия могут быть идемпотентными. Если правильно спроектировать бизнес-сущности, операция станет идемпотентной. Представьте сервис покупательской корзины, отправляющий события со статусом продукта на сайт интернет-магазина. Такие события могут потребляться другими сервисами, для которых содержимое корзины представляет интерес.

В этом разделе мы спроектируем событие в двух вариантах: одно будет идемпотентным, а другое неидемпотентным. Самый прямолинейный подход — отправка события, сообщающего о добавлении товара в корзину при каждом добавлении. Например, если пользователь добавляет в корзину новую книгу А, отправляется новое событие с количеством 1. Если пользователь снова поместит в корзину ту же книгу, отправится событие с количеством 2 (рис. 10.4).

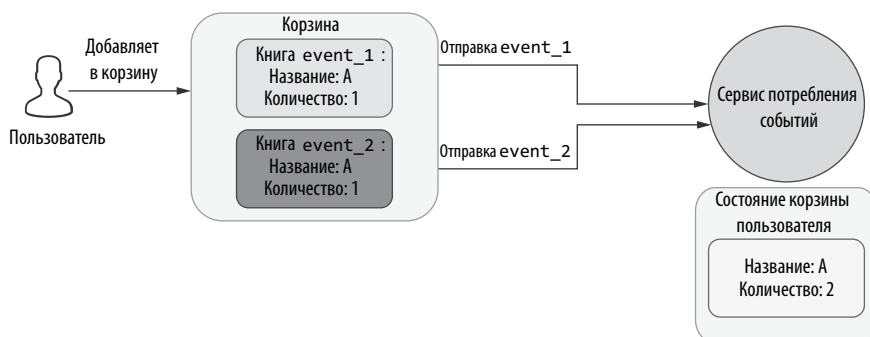


Рис. 10.4. Неидемпотентные операции, производящие данные в корзине для сервиса продажи книг

Сервис-потребитель событий книг строит собственное представление состояния корзины на основании отправленных событий. Архитектура на базе событий часто используется для построения системы на основе паттерна CQRS (Command Query Responsibility Segregation — «разделение обязанностей команд и запросов»). Это позволяет независимо масштабировать операции записи (сервис корзины) и операции чтения системы. В нашем сценарии события от сервиса корзины отправляются в очередь и используются несколькими независимыми сервисами-потребителями. Каждый сервис может построить свою модель базы данных, оптимизируя ее для своего трафика чтения. Кроме того, добавление других сервисов на стороне чтения не влияет на производительность записи сервиса корзины. (Этот паттерн более подробно рассматривается в следующем разделе.)

Проблема представленной бизнес-модели в том, что она не идемпотентна. Если сервис корзины должен повторять попытку отправки для каждого события корзины, сервису-потребителю событий книг будет доставляться дублируемое состояние. Так как сервис-потребитель должен воссоздать представление корзины на основании событий, он будет увеличивать количество единиц товара для каждого дублируемого события. Итоговое количество будет равно 3. Целостность представления нарушается. Очевидно, такая бизнес-модель идемпотентной не является. Как переработать ее, чтобы это исправить?

Вместо отправки события с каждым изменением сервис корзины может отправить событие с полным представлением корзины. В улучшенной архитектуре каждый раз при добавлении новой позиции в корзину внешнему сервису отправляется новое агрегированное событие. Первый раз, когда пользователь добавляет в корзину книгу А, отправляется событие с количеством 1. Но при повторной отправке книги А новое событие будет содержать количество, равное 2. Из-за этого все сервисы-потребители событий корзины получают полное представление корзины. Им не нужно воссоздавать локальное представление, которое может утратить целостность при повторных попытках. Сервис корзины повторно отправляет события без риска нарушения целостности состояния.

Существует только одна ловушка, о которой следует знать. В случае повторной попытки сервис корзины все еще может выдать дубликат. Так как с событиями распространяется полное состояние корзины, более новое событие, отправленное клиентам, может переопределить старое состояние корзины для пользователя. Однако в случае повторных попыток возможно нарушение порядка событий, как показано на рис. 10.5.

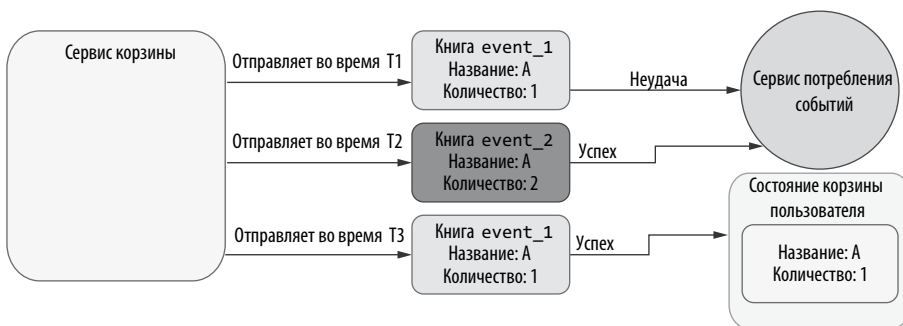


Рис. 10.5. Нарушение порядка повторных попыток

Допустим, первая операция в момент $T1$ завершается неудачей и повторная попытка планируется на заданное время в будущем. Более поздняя операция в момент $T2$ завершается успешно. Затем в момент $T3$ выполняется

запланированная повторная операция, и она переопределяет состояние, распространяемое событием `Book event_1`. Сервис-потребитель событий завершается с нарушением целостности состояния. Из-за этого необходимо внимательно подходить к повторным попыткам того же пользователя. Проблема решается упорядочением событий на стороне клиента или упорядочением событий, отправляемых на стороне сервиса корзины. Также не следует нарушать упорядочение повторными попытками.

Как правило, глобальное упорядочение событий не обязательно: корзина создается конкретным пользователем и принадлежит ему; каждый из них получает уникальный идентификатор. Следовательно, мы можем распространять `user_id` владельца корзины. С такой информацией достаточно упорядочивать события только для этого конкретного идентификатора. Если события корзины упорядочиваются в рамках `user_id`, сервисы, потребляющие события, могут воссоздать корзину для `user_id`, не особенно беспокоясь о переопределении поведения. Можно сказать, что данные корзины секционируются по `user_id`, и упорядочение гарантировано в пределах раздела. Широко используемые фреймворки очередей (например, Apache Kafka и Pulsar) предоставляют средства для обеспечения порядка внутри раздела.

Распространение полного состояния представления также имеет ряд недостатков. Если состояние увеличивается, придется передавать больше данных по сети при каждой отправке события. Также это означает, что логика сериализации и десериализации должна выполнять больше работы. Впрочем, в реальных системах идемпотентность бизнес-модели часто оправдывает эти компромиссы.

Как видно из примера, наделение свойством идемпотентности операций (кроме чтения) — дело сложное, нестабильное, а иногда даже невозможное. В распределенной архитектуре со множеством компонентов (например, CQRS) проблемы только умножаются.

10.1.4. Паттерн CQRS

Чтобы лучше понять суть CQRS, предположим, что мы хотим построить два сервиса, потребляющие данные корзины пользователей. Существующий сервис корзины отвечает за запись события пользователя в долгосрочную очередь. Это компонент команд (C) модели записи нашей архитектуры. С другой стороны, события пользователей могут асинхронно (когда-нибудь в будущем) потребляться *N* сервисами. Будем считать, что для этого есть два сервиса: пользовательских профилей и реляционного анализа (рис. 10.6).

Сервис пользовательских профилей должен оптимизировать свою модель чтения для ускорения получения данных по `user_id`. Можно выбрать распределенную базу данных и применять `user_id` в качестве ключа секционирования.

Затем клиенты пользовательского профиля могут запрашивать информацию у сервиса по `user_id` с помощью модели данных, оптимизированной для чтения. Вторая модель данных у сервиса реляционного анализа оптимизируется для других сценариев использования. Она тоже читает данные пользователей, но строит совершенно иную модель чтения, оптимизированную для автономного анализа, и позволяет применять разные паттерны, оптимизированные для пакетных запросов. События могут сохраняться в распределенной файловой системе (скажем, HDFS). Оба сервиса представляют компонент запросов (Q) архитектуры CQRS.

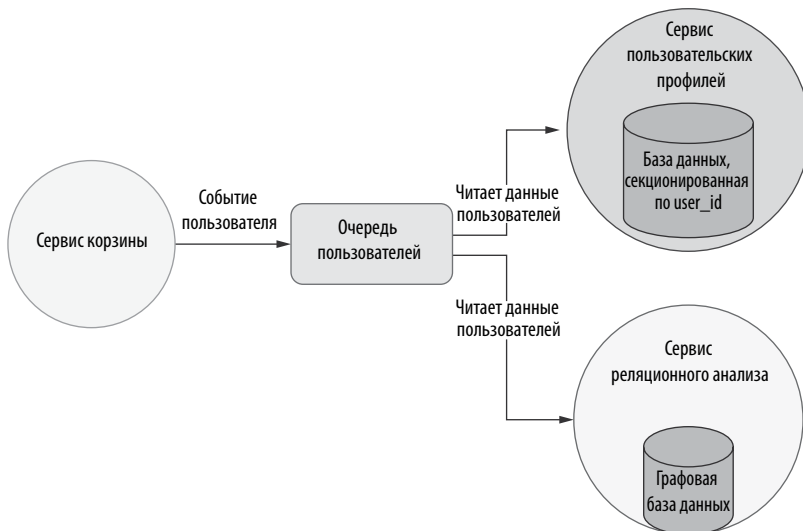


Рис. 10.6. Применение CQRS с двумя моделями чтения

У этой архитектуры есть несколько важных преимуществ. Во-первых, производители и потребители данных отделены друг от друга. Во-вторых, сервис, производящий события, не обязан предугадывать все возможные будущие способы применения данных. Он сохраняет события в хранилище данных, оптимизированном для записи. Потребитель обязан загрузить эти данные и преобразовать их в модель базы данных, оптимизированную для конкретного сценария использования. Команды, разрабатывающие сервисы-потребители, могут работать независимо, создавая ценность для бизнеса на основании общедоступных данных. При использовании CQRS данные выходят на первый план. Сервисы-потребители могут потреблять разные источники данных и использовать их для своих целей.

Тем не менее у этого паттерна есть и недостатки. Во-первых, данные будут дублироваться в N местах. Чем больше понадобится сервисов с моделями

чтения, тем больше объем дублирования. Кроме того, эта архитектура требует значительного перемещения данных. В ней будет множество запросов, отправляемых как из сервисов моделей записи (для сохранения исходных данных), так и из сервисов моделей чтения. Любые из этих запросов могут завершиться неудачей, так что все упомянутые ранее проблемы (повторные попытки, доставка «не менее одного раза», нарушение связности сети, идемпотентность операции) повлияют на состояние системы. Собственно, они станут еще более выраженными: чем больше сервисов, тем выше риск возникновения проблем. Между сервисами моделей чтения может возникнуть рассинхронизация состояния, если вы не защититесь от этого должным образом. Один неидемпотентный дубликат, отправленный одному из двух сервисов, может привести к расхождению состояния всей системы.

Как спроектировать отказоустойчивую систему (то есть повторяющую неудачные действия), которая работает в распределенной среде (на практике это почти любая система, находящаяся в реальной эксплуатации), и гарантировать целостность представления системы? Проверенный паттерн для подобных задач — логика дедупликации, реализованная на стороне потребителя. Когда сервис, выполняющий неидемпотентное действие (то, которое невозможно просто повторить), реализует логику дедупликации, он фактически изменяет свое поведение, чтобы оно было идемпотентным для всех вызывающих сторон. В следующем разделе мы реализуем такую логику в библиотеке.

10.2. НАИВНАЯ РЕАЛИЗАЦИЯ ДЕДУПЛИКАЦИИ

Попробуем сделать рассылку сообщений почтовым сервисом идемпотентной. Это можно сделать, реализовав в сервисе логику дедупликации. При поступлении нового запроса сервис проверяет, был ли он доставлен ранее. Если запрос не был доставлен, значит, это не дубликат и его можно безопасно обработать.

Важно отметить, что для работы дедупликации каждому событию нужно присвоить уникальный идентификатор. Вызывающий сервис (приложение А) генерирует UUID-идентификатор, однозначно определяющий каждый запрос. При повторной отправке запроса используется тот же UUID. С этой информацией почтовый сервис, получающий событие, проверяет, поступало ли оно ранее. Если в архитектуре запрос (или событие) может перемещаться между несколькими сервисами, все они могут использовать один уникальный идентификатор запроса. Обычно идентификатор генерируется на стороне производителя (первый сервис, выполняющий запрос или событие) и может использоваться для дедупликации несколькими сервисами на своем пути.

Информация о том, был ли обработан идентификатор, должна храниться в течение долгого времени. Поэтому идентификатор находится в базе данных, обеспечивающей постоянное хранение. База данных — новый компонент, который должен использоваться в системе. Весьма вероятно, что сервис уже использует базу данных, так что можно просто добавить новую специальную таблицу для дедупликации. Логика дедупликации представлена на рис. 10.7.



Рис. 10.7. Логика дедупликации в почтовом сервисе

Рассмотрим ту же ситуацию, которая приводила к появлению дубликатов сообщений. Отправляется первый запрос (на рис. 10.7) с идентификатором 1234 (в реальном приложении это будет UUID). Получив его, почтовый сервис проверяет, обрабатывался ли уже запрос с заданным идентификатором. Для этого выполняется запрос к базе данных. Если обработанного события с заданным идентификатором не было, запись добавляется в базу данных. Следующий шаг — отправка сообщения конечному пользователю. Затем почтовый сервис отправляет информацию (шаг 4) о том, что данные были обработаны успешно, но внезапно происходит нарушение связности сети (шаг 5).

Приложение A не знает, отправлено сообщение или нет, поэтому оно повторяет запрос с тем же идентификатором. Когда повторный запрос поступает к почтовому сервису, тот проверяет, является ли запрос дубликатом. Если запрос уже был обработан, то он не обрабатывается заново.

Решение выглядит надежно, но у него есть недостаток. Что произойдет, если сбой возникнет после сохранения идентификатора обработанного запроса почтовым сервисом, но до фактической отправки сообщения? Ситуация показана на рис. 10.8.

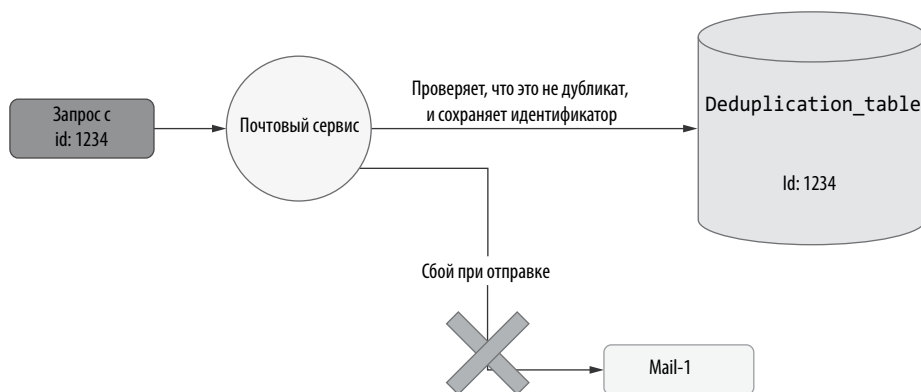


Рис. 10.8. Частичный сбой при отправке

Если сервис дедупликации проверяет и сохраняет идентификатор события перед отправкой, возникает риск частичного сбоя. Допустим, после того как запрос был помечен как обработанный, в процессе отправки почты происходит сбой. Ответ со сбоем отправляется вызывающему приложению. Как и ожидалось, оно делает повторную попытку с тем же идентификатором запроса. Однако почтовый сервис уже пометил заданный идентификатор как обработанный. Таким образом, запрос не обрабатывается и письмо не отправляется. Самое очевидное решение — разбивка сервиса дедупликации на два этапа и вставка действия отправки почты между ними. Процесс показан на рис. 10.9.

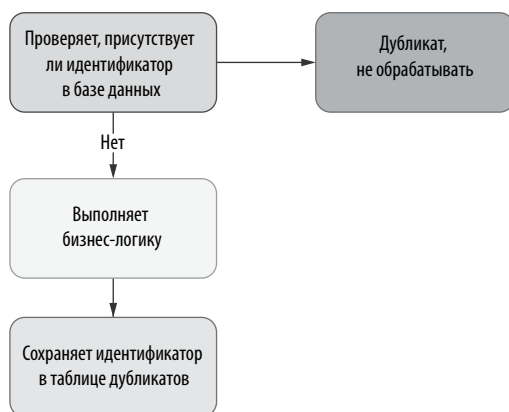


Рис. 10.9. Дедупликация из трех этапов

Сначала новое решение пытается прочитать из базы данных запись с заданным идентификатором. Если идентификатор отсутствует в базе, выполняется действие, предоставляемое вызывающей стороной. В нашем сценарии это отправка почты. После того как отправка завершится успешно (вернет управление без

исключения), можно будет добавить новую запись с идентификатором запроса. Реализация этой логики приведена в листинге 10.1.

Листинг 10.1. Наивная реализация дедупликации

```
public class NaiveDeduplicationService {

    private final DbClient dbClient = new DbClient();

    public void executeIfNotDuplicate(String id, Runnable action) {
        boolean present = dbClient.find(id);
        if (!present) {
            action.run();
            dbClient.save(id);
        }
    }
}
```

`DbClient` отвечает за взаимодействие с базой данных. Предоставленный объект `Runnable` — наш процесс отправки сообщения. Вызов `dbClient.find(id)` представляет первый этап обработки. Он пытается определить, присутствует ли запись в базе данных. Если ее нет, выполняется фактическая обработка. Последний этап сохраняет запись с новым идентификатором в базе данных. Если запись с идентификатором уже присутствует в базе, запрос игнорируется.

Похоже, это решение плохо работает в обоих рассматриваемых сценариях со сбоями. Если после успешной отправки письма нарушается связность сети, идентификатор запроса уже хранится в базе данных (после вызова метода `dbClient.save()`). В этом случае повторные запросы будут определяться как дубликаты.

Во втором рассматриваемом сценарии (при сбое во время отправки сообщения) происходит отказ во время обработки `Runnable`. Из-за этого идентификатор запроса не сохраняется в базе данных. При повторной попытке выполнения запрос будет нормально обработан заново, потому что идентификатор не сохранен.

Однако стоит помнить, что почтовый сервис работает в распределенной среде. Так как она изначально подразумевает параллельное выполнение, обсуждаемое решение не обеспечит идемпотентность всех сценариев использования. Давайте разберемся, почему решение не атомарно и как реализовать его с соблюдением атомарности.

10.3. ТИПИЧНЫЕ ОШИБКИ ПРИ РЕАЛИЗАЦИИ ДЕДУПЛИКАЦИИ В РАСПРЕДЕЛЕННЫХ СИСТЕМАХ

Рассмотрим наивную реализацию из предыдущего раздела в двух контекстах. Первый из них предполагает, что почтовый сервис и приложение А, которое отправляет ему данные, развертываются только на одном узле. Второй контекст добавляет сложность: почтовый сервис развертывается на нескольких

узлах. Этот сценарий использования более реалистичен, потому что именно так сервисы реализуются в микросервисной архитектуре, отказоустойчивой и масштабируемой. Наличие нескольких узлов обеспечивает отказоустойчивость, потому что в случае сбоя одного узла другой узел (или узлы) начинает обрабатывать его трафик. Проанализируем, как контекст влияет на целостность логики дедупликации.

10.3.1. Одноузловый контекст

Посмотрим, как логика дедупликации работает в контексте одного сервиса приложения А и одного почтового сервиса. Оба сервиса развертываются только на одном узле. На рис. 10.10 представлен этот контекст.

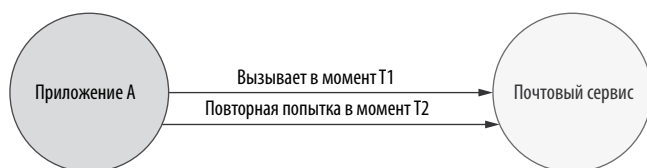


Рис. 10.10. Одноузловый контекст для приложения А и почтового сервиса

Проанализируем повторную попытку для заданного уникального идентификатора. Первый вызов, выполняемый приложением А, происходит в момент времени 1 ($T1$). Допустим, он завершается неудачей, и повторная попытка делается после его сбоя в момент времени 2 ($T2$). Мы снова будем считать, что первый запрос связан с действием повторной попытки отношением «происходит ранее». В нашем случае текущая логика дедупликации не атомарна. Она разбита на три этапа:

- *Этап 1* — ищет `request-id` в базе данных.
- *Этап 2* — выполняет логику почтового сервиса, если идентификатор `request-id` не найден.
- *Этап 3* — сохраняет `request-id` в базе данных.

Для простоты ограничимся сбоем на этапе 2, но в реальном приложении сбой может произойти на любом этапе, что дополнительно усложняет анализ задачи. Сосредоточимся на главной функции компонента: предотвращении дублирования отправки сообщений.

Если у первого запроса ($T1$) происходит сбой на этапе 2, вызывающему приложению возвращается ответ. Из-за сбоя на этапе 2 действие этапа 3 не выполняется. Выполняется повторная попытка в точке $T2$, на этот раз действие отправки выполняется успешно. Возможность отправки дубликата в этом случае исключается, даже если метод `executeIfNotDuplicate()`, приведенный в следующем листинге, не является атомарным.

Листинг 10.2. Блокирование действий отправки сообщений

```

public void executeIfNotDuplicate(String id, Runnable action) {
    boolean present = dbClient.find(id);
    if (!present) {
        action.run();
        dbClient.save(id);
    }
}

```

← Блокирует действия
отправки на N секунд

Посмотрим, что произойдет, если действие отправки занимает много времени. В листинге действие отправки является блокирующим и подразумевает еще один удаленный вызов (фактическая отправка сообщения), способный блокировать обработку кода. Сбой также может произойти во время ответа из-за нарушения связности сети; возникает ситуация, описанная в разделе 10.1.2, но на этот раз она применяется к внешнему вызову по сети.

Как вы знаете из главы 9, для каждого сетевого запроса должен определяться разумный тайм-аут, предотвращающий блокирование потоков и ресурсов. Допустим, приложение А определяет тайм-аут продолжительностью в 10 с, но отправка сообщений почтовым сервисом блокируется на вдвое большее время (20 с). В этом случае сбой запроса в момент $T1$ происходит через 10 с. Тем не менее это не означает, что отправка сообщения завершилась неудачей. Возможно, все пройдет успешно, только через 10 с. Ситуация изображена на рис. 10.11.

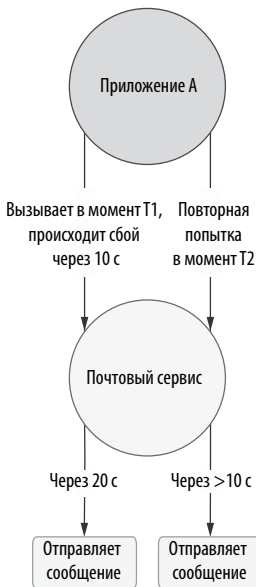


Рис. 10.11. Контекст одного сервиса с дублированием отправки

Выполнение запросов чередуется. С точки зрения приложения А у первого запроса в точке *T1* происходит сбой (по тайм-ауту). Однако основное действие блокируется на 20 с и по прошествии этого времени завершается успешно. Затем приложение просто сохраняет свой идентификатор запроса в базе данных. К сожалению, за это время приложение А выдает повторную попытку запроса в точке *T2*, потому что оно столкнулось со сбоем. Повторный запрос поступает почтовому сервису до того, как он сохранит `request-id` из *T1* как уже обработанный запрос. Так как дубликат был отправлен, наш неатомарный сервис дедупликации нарушает целостность представления системы.

Это только один из сценариев со сбоями, которые могут создать дубликат в одноузловом контексте. Тем не менее при проектировании надежного компонента достаточно даже одного сценария использования, в котором нарушаются требования, чтобы пересмотреть решение. Но сначала проанализируем тот же сценарий в многоузловом контексте.

10.3.2. Многоузловый контекст

Чтобы проанализировать целостность и правильность логики дедупликации в многоузловом контексте, рассмотрим ситуацию, в которой почтовый сервис развертывается на ряде узлов. Развертывание сервиса на нескольких физических машинах (узлах) в целях повышения его общей производительности и отказоустойчивости применяется довольно часто.

Когда почтовый сервис развертывается на нескольких узлах, его API предоставляется через балансировщик нагрузки. Каждый сервис доступен по IP-адресу. Будем считать, что эта схема обеспечивает динамическую масштабируемость; это означает, что новые экземпляры почтового сервиса могут добавляться или удаляться в зависимости от трафика. Из-за этого IP-адреса экземпляров почтового сервиса скрываются от приложения А. Запрос, выполняемый приложением А, отправляется сервису балансировки нагрузки, который перехватывает запрос и перенаправляет его конкретному экземпляру почтового сервиса.

Фактическая реализация распределения нагрузки абстрагируется от сервиса приложения А. Новый развертываемый почтовый сервис регистрируется в сервисе балансировки нагрузки. С этого момента сервис балансировки нагрузки начинает маршрутизировать трафик добавленному узлу. На рис. 10.12 изображена роль балансировщика нагрузки в контексте с несколькими узлами.

В этом сценарии почтовый сервис не должен обладать состоянием; он должен быть способен обработать любой входящий запрос. Все необходимое состояние, включая таблицу уже обработанных `request-id`, хранится в отдельной базе данных. Для простоты анализа предположим, что база данных не распределена, а все состояние хранится в одном узле. Однако в реальности масштабируемое

приложение (это свойство достигается добавлением или удалением узлов), вероятно, должно использовать распределенную базу данных, так что данные `request-id` секционируются по N узлам. Это также открывает возможность горизонтального масштабирования уровня данных посредством добавления и удаления узлов. Тем не менее сценарии со сбоями, которые мы рассматриваем, будут присутствовать при использовании баз данных обоих типов (распределенных и нераспределенных).

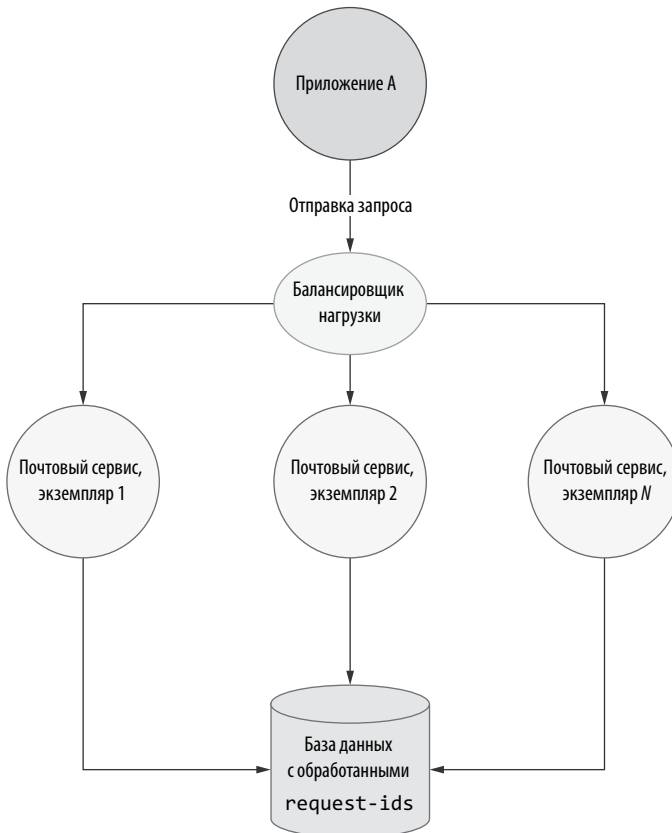


Рис. 10.12. Контекст с одним сервисом

Допустим, компонент балансировщик нагрузки выбирает экземпляр почтового сервиса по простой циклической схеме. Первый запрос направляется почтовому сервису 1, второй — почтовому сервису 2 и т. д. Алгоритмы балансировки нагрузки широко применяют циклическую схему: она понятна и просто реализуется, к тому же эффективна на практике. Существуют и алгоритмы балансировки нагрузки, которые, например, учитывают задержку узлов. Один из самых популярных методов такого рода — алгоритм выбора по степеням двойки (<http://mng>).

bz/DxPR). Впрочем, конкретный алгоритм, используемый сервисом балансировки нагрузки, не влияет на анализ.

К сожалению, текущая логика дедупликации не будет правильно работать в такой среде. Рассмотрим сценарий, в котором приложение А повторяет запрос в многоузловом контексте, как показано на рис. 10.13.

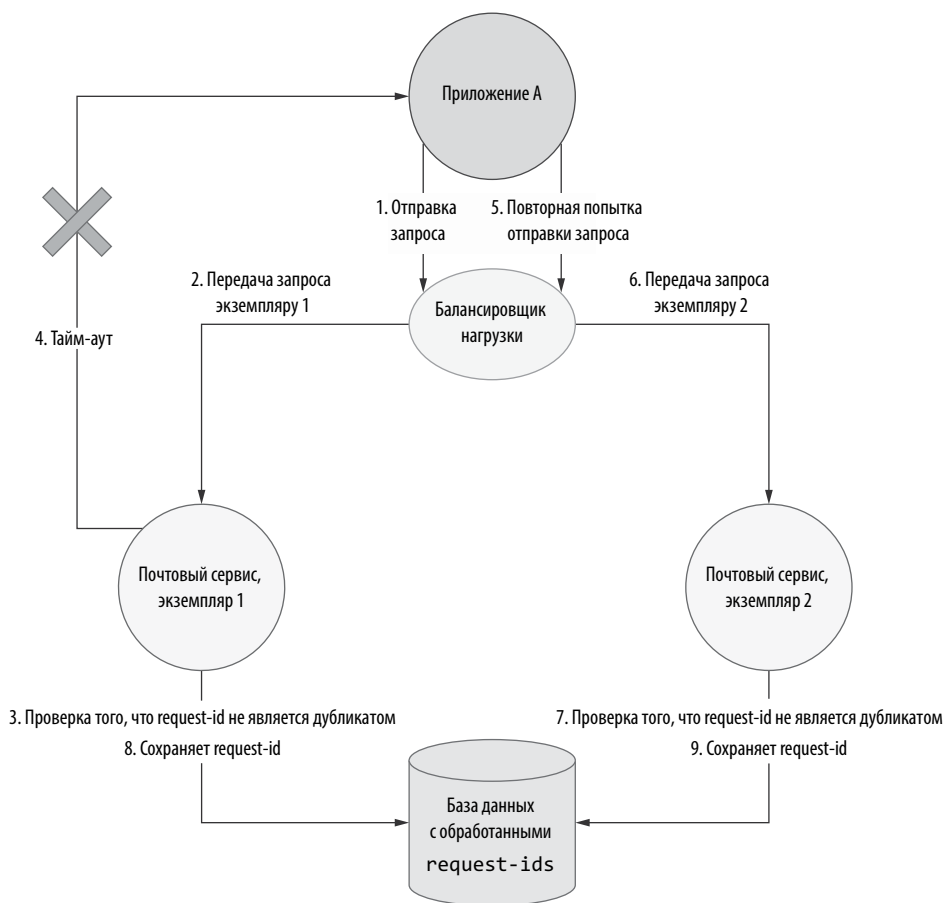


Рис. 10.13. Повторные попытки отправки запросов в контексте с несколькими сервисами

На шаге 1 приложение А отправляет запрос на отправку сообщения. Запрос проходит через балансировщик нагрузки, и на шаге 2 передается первому экземпляру почтового сервиса. На шаге 3 почтовый сервис проверяет, присутствует ли `request-id` в базе данных. Его там нет, поэтому обработка продолжается. К сожалению, этот шаг приводит к тайм-ауту, который возвращается приложению А

на шаге 4. Приложение А выдает повторную попытку на шаге 5, и этот запрос повторной попытки передается второму экземпляру почтового сервиса на шаге 6. На шаге 7 почтовый сервис проверяет, был ли идентификатор запроса обработан ранее. Если выяснится, что он не был обработан, выполняется отправка. Тем временем первый экземпляр почтового сервиса завершает запрос на отставку сообщения и на шаге 8 сохраняет идентификатор запроса в базе данных. Затем на шаге 9 второй экземпляр завершает свое выполнение и сохраняет идентификатор запроса в базе данных, переопределяя предыдущую операцию сохранения, которая была выдана первым экземпляром. Это также означает, что оба экземпляра почтового сервиса не увидели дубликат в начале логики дедупликации, что привело к повторному выполнению логики фактической отправки сообщения.

В реальной ситуации все может быть еще хуже. Приложение А инициирует отставку сообщения на основании некоторой логики. Может оказаться, что логика инициируется еще одним внешним вызовом от другого сервиса. В микросервисной архитектуре (особенно на основе событий) подобная ситуация встречается не так редко. Бизнес-обработка может охватывать несколько сервисов. Кроме того, если предположить, что приложения не имеют состояния, приложение А также может получить дубликаты запросов. По этой причине логика дедупликации не атомарна и может привести к еще большему дублированию. Это сильно повлияет на целостность представления системы, потому что не исключено появление большого количества дубликатов. На этом этапе анализа становится ясно, что логика дедупликации нуждается в усовершенствовании. Посмотрим, как обеспечить ее атомарность в контекстах с одним и несколькими узлами.

10.4. ОБЕСПЕЧЕНИЕ АТОМАРНОСТИ ЛОГИКИ ДЛЯ ПРЕДОТВРАЩЕНИЯ СИТУАЦИИ ГОНКИ

Вспомним текущую логику дедупликации. Она включает три этапа:

- *Этап 1* — ищет `request-id` в базе данных.
- *Этап 2* — выполняет логику почтового сервиса, если идентификатор `request-id` не найден.
- *Этап 3* — сохраняет `request-id` в базе данных.

Стоит отметить, что все рассмотренные сценарии со сбоем нарушали целостность представления системы независимо от того, присутствует этап 2 в логике или нет. Упростим пример и будем считать, что логика дедупликации включает только этапы 1 и 3. Тогда она принимает следующий вид:

- *Этап 1* — ищет `request-id` в базе данных.
- *Этап 2* — сохраняет `request-id` в базе данных.

Вероятность отправки дубликата сообщения все еще остается, потому что вызовы чтения из базы данных и сохранения информации в ней также могут завершиться неудачей — ведь это удаленные вызовы, которые выполняются в распределенной системе. Также возможно нарушение связности сети при отправке успешного ответа от базы данных через логику дедупликации.

Все сценарии сбоев, описанные в контексте приложения А, применимы и к вызовам базы данных. Например, при вызове операции сохранения `request-id` (этап 3) операция может выдать исключение — признак тайм-аута. Как известно, тайм-аут не предоставляет вызывающей стороне подробной информации. Возможны ситуации, в которых происходит тайм-аут на стороне клиента, но операция на стороне сервера продолжает выполняться. С точки зрения приложения А это означает, что действие завершается неудачно, а клиенту возвращается ошибка. Повторная попытка может случиться до того, как значение `request-id` будет добавлено в таблицу экземпляром сервиса 1. Таким образом, запрос направится второму экземпляру сервиса. Ситуация почти такая же, как обсуждалась в предыдущем разделе. На рис. 10.14 показано, как это может привести к ситуации гонки (race condition).

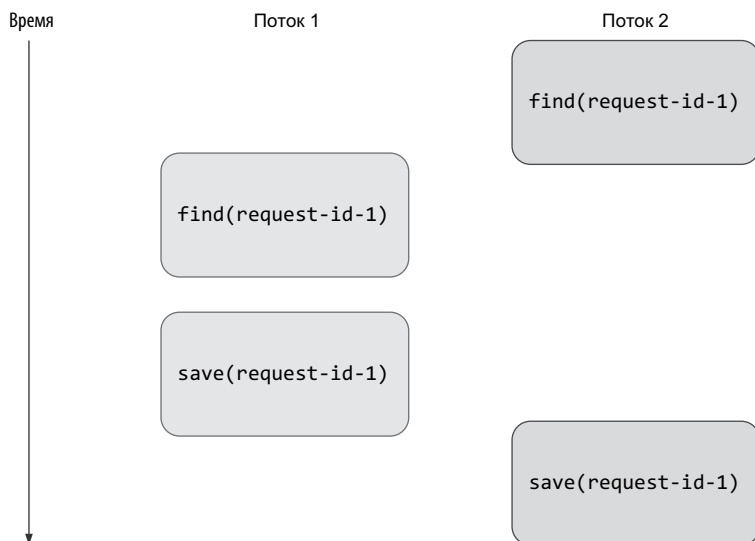


Рис. 10.14. Поиск и сохранение в двух операциях приводит к ситуации гонки

Действия поиска и сохранения могут чередоваться, что приводит к нарушению целостности системы. Например, операция поиска в одном потоке (или узле) может быть выполнена уже после ее выполнения другим узлом и занять неопределенное количество времени. Значит, никаких сильных предположений здесь сделать нельзя. Для обоих вызовов поиска будет возвращено значение `false`, логика продолжит выполнение, и наконец, операция сохранения будет вызвана дважды. Из-за этого логика дедупликации работает неправильно.

Чтобы добиться атомарности логики дедупликации, нужно сократить количество этапов до одного. Также необходимо проверить, является ли заданный запрос дубликатом, и сохранить `request-id` за одну операцию. Должен быть один внешний вызов без промежуточных этапов. Каждый раз, когда запросу нужно получить значение, выполнить некоторое действие и сохранить другое значение, возникает потенциальный риск ситуации гонки.

Это справедливо при выполнении в многопоточной среде. Можно синхронизировать все вызовы компонента дедупликации, но это означает, что уровень параллелизма компонента будет равен 1. Иначе говоря, сервис будет обрабатывать запросы по одному. Такие решения не годятся для реальных приложений, которые должны обрабатывать N запросов в секунду. Чем больше запросов должна обрабатывать система, тем выше параллелизм и количество потоков. Это увеличивает вероятность промежуточных сбоев, из-за которых логика дедупликации быстрее теряет целостность.

К счастью, многие распределенные базы данных, которые чаще всего используются в горизонтально масштабируемой архитектуре, дают возможность выполнения задач в виде одиночных атомарных операций. (Стандартная база данных SQL также позволяет выполнять атомарные операции.)

Требуется выполнить операцию сохранения, которая добавляет новую запись только в том случае, если она еще не присутствует в базе данных. Более того, она должна возвращать логический признак того, успешно ли прошла вставка. Такая операция предоставляет всю информацию, необходимую для реализации надежной логики дедупликации. Это называется *обновлением/вставкой* (upsert); действие сохранения вставляет значение, только если оно еще не присутствует в базе, и возвращает результат. Концепция обновления/вставки показана на рис. 10.15. Следует узнать, поддерживает ли ее выбранная база данных. Операция обновления/вставки должна быть атомарной, то есть база данных должна выполнить ее как одну операцию.

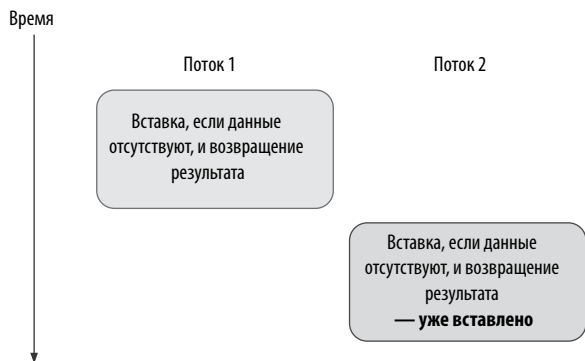


Рис. 10.15. Атомарная операция обновления/вставки добавляет значение только в том случае, если оно еще не присутствует в базе, и возвращает результат

Так как операция обновления/вставки выполняется как атомарная, между двумя чередующимися операциями не возникает ситуации гонки. Вся логика выполняется на стороне базы данных, и результат возвращается вызывающей стороне.

Посмотрим, как изменяется реализация сервиса дедупликации при использовании обновления/вставки. Здесь важнее всего то, что `DbClient` предоставляет метод, который позволяет вставить данные при их отсутствии, и возвращает результат. Метод приведен в листинге 10.3.

Листинг 10.3. Применение обновления/вставки для дедупликации

```
public boolean findAndInsertIfNeeded(String id);
```

Необходимо убедиться, что используемая реализация является атомарной. Для этого попытаемся добавить запись с заданным идентификатором. При наличии записи в базе данных возвращается `false`. Если была выполнена вставка, возвращается `true`. В следующем листинге приведен новый метод `isDuplicated()`, который возвращает `true` или `false` в зависимости от того, является ли заданный идентификатор дубликатом.

Листинг 10.4. Реализация атомарной логики дедупликации

```
@Override
public boolean isDuplicate(String id) {
    boolean wasInserted = dbClient.findAndInsertIfNeeded(id);
    if (wasInserted) {
        return false;
    } else {
        return true;
    }
}
```

Если `findAndInsertIfNeeded()` возвращает `true`, это означает, что заданный идентификатор был вставлен в базу данных, то есть до сих пор его там не было. Здесь важно то, что метод добавляет заданный идентификатор в базу данных. Теперь этап 2, необходимый до этого, не нужен. Если метод `findAndInsertIfNeeded()` возвращает `false`, идентификатор является дубликатом. Также это значит, что операция обновления/вставки не добавила новый идентификатор, поскольку значение уже присутствовало.

Теперь логика стала атомарной, следовательно, нет риска попасть в ситуацию гонки. Заметим, что использование атомарной операции, которая одновременно выполняет вставку и проверяет, присутствует ли значение, не позволяет произвести нестандартное действие между этими этапами. Но вы уже увидели, что такой подход неверен. Новая логика дедупликации отвечает только за поиск дубликатов. Она не пытается проверить успешность сквозного выполнения запроса и имеет только одну функцию, которая выполняется правильно и атомарно.

При использовании нового компонента дедупликации в почтовом сервисе без механизма, проверяющего правильность отправки сообщения, появляется риск того, что письмо не доставлено. Рассмотрим ситуацию, в которой логика дедупликации помечает запрос как обработанный в тот момент, когда запрос поступает в почтовую систему. Если сбой обработки произойдет после этого, повторная попытка запроса будет проигнорирована, потому что запрос помечен как обработанный.

С другой стороны, если дубликат помечается после успешной обработки, не существует механизма, предотвращающего отправку дубликата. Вот почему стоит использовать атомарную дедупликацию при входе в систему. Ее следует применять совместно с другими механизмами, которые проверяют правильность работы системы в случае сбоя — такими, как журналы транзакций или откат (отмена) обработанных идентификаторов. У всех этих техник свои сложности и компромиссы, и их следует анализировать отдельно.

Пример из этой главы демонстрирует, что выполнение и анализ действий в распределенной системе — дело сложное и запутанное. Если вы сможете спроектировать идемпотентную обработку, система станет более отказоустойчивой и надежной. Однако не все сервисы обработки могут быть идемпотентными, поэтому необходимо спроектировать механизм, защищающий систему от выполнения повторных попыток действий, для которых они недопустимы. Если вы не хотите проектировать сложную логику дедупликации, каждый сбой запроса будет критичным для приложения, потому что вы не сможете выполнить повторную попытку. Только ручное вмешательство системного администратора может обеспечить согласование данных. Если вы хотите, чтобы система действительно была отказоустойчивой и надежной, такой подход не лучшее решение.

По этой причине для устранения подобных проблем можно реализовать такие механизмы, как дедупликация с повторными попытками. Однако нужно проявить осторожность, потому что реализация таких механизмов в распределенных системах может придать системе не те характеристики, на которые вы рассчитывали. Опасно создавать систему, которая должна быть целостной, но выполняет разнородные операции — возникает риск появления трудноустраняемых ошибок и потери средств из-за выполнения дубликатов транзакций. Поэтому следует проанализировать весь входящий и исходящий трафик в контексте правильности и семантики доставки. В следующей главе подробнее рассматриваются семантика доставки и поток данных между приложениями.

ИТОГИ

- Если ваше приложение выполняет любые сетевые вызовы, вы работаете в распределенной среде. Помните, что каждый такой вызов может завершиться сбоем.

- Сбои внешних вызовов могут объясняться разными причинами: сетевыми отказами, сбоями целевого приложения и т. д. Тем не менее такие сбои можно анализировать и делать выводы.
- Механизмы повторных попыток позволяют проектировать отказоустойчивые приложения.
- Идемпотентные операции позволяют совершать повторные попытки, не беспокоясь о возможном дублировании.
- Бизнес-область можно спроектировать так, чтобы она была более склонной к идемпотентности. Чем больше операций являются идемпотентными, тем более автономной и отказоустойчивой будет система.
- Помимо идемпотентности, следует тщательно отслеживать порядок запросов. Проанализируйте, как идемпотентность влияет на стратегии повторных попыток, применяемые в приложении.
- При реализации логики, работающей в распределенном контексте (например, в библиотеке дедупликации), необходимо тщательно анализировать граничные случаи и сценарии сбоев.
- Обеспечить атомарность операций в системе бывает сложно, а иногда и невозможно, если обработка, которая должна быть атомарной, делится на N этапов. Неатомарное решение можно превратить в атомарное с помощью подходящих операций с базой данных.
- При разделении действия, которое должно быть целостным, на N удаленных вызовов возникает риск нарушения целостности системы.
- Во всех системах существуют гарантии, которые можно использовать в коде. Если взаимодействие между ними требует внешнего вызова, каждый из таких вызовов может завершиться сбоем.
- При использовании системы, предназначенной для работы в распределенной среде, решаемая проблема с довольно высокой вероятностью уже была решена. Например, многие атомарные операции, которые на первый взгляд трудно реализовать вручную, реализуются методом обновления/вставки. Это улучшает целостность системы.

11

Семантика доставки в распределенных системах

В этой главе

- ✓ Модели «публикация — подписка» и «производитель — потребитель» в приложениях с обработкой больших объемов данных.
- ✓ Гарантии доставки и их влияние на отказоустойчивость.
- ✓ Построение отказоустойчивых систем с применением семантики доставки.

В главе 10 вы узнали об отказоустойчивости, повторных попытках и идемпотентности операций в контексте относительно прямолинейной архитектуры систем. Реальные системы представляют собой набор компонентов, отвечающих за разные части бизнес-области и инфраструктуры. Например, один сервис может отвечать за сбор метрик, другой — за ведение журнала сбора и т. д. Кроме того, нужны приложения, реализующие первичные бизнес-сценарии для предметной области — например, платежный сервис или база данных, отвечающая за хранение информации. В таких архитектурах сервисы должны связываться друг с другом, чтобы обмениваться информацией.

Чем больше компонентов в системе, тем больше точек, в которых может произойти сбой. Он может возникнуть при каждом сетевом запросе, и необходимо определить, нужно ли повторять операцию. Если вы хотите создать отказоустойчивую архитектуру, необходимо встроить в систему обработку сбоев. Тогда каждый компонент должен обеспечить точную семантику доставки при

производстве данных. С другой стороны, потребление данных тоже должно следовать ожидаемой семантике доставки.

В этой главе вы узнаете, как строить событийно-управляемые архитектуры, чтобы создавать отказоустойчивые системы со слабой связанностью. В качестве главного компонента системы будет использоваться Apache Kafka. Это позволит изучить на практике различные варианты семантики: не более одного, (фактически) ровно один, не менее одного. Наконец, мы встроим отказоустойчивость в систему, чтобы обеспечить ожидаемые гарантии доставки. Начнем с разбора событийных архитектур приложений с обработкой больших объемов данных и рассмотрим их достоинства и недостатки.

11.1. АРХИТЕКТУРА СОБЫТИЙНО-УПРАВЛЯЕМЫХ ПРИЛОЖЕНИЙ

Зачем прикладывать усилия для реализации системы с событийно-управляемой архитектурой? Начнем с простого решения и посмотрим, как оно эволюционирует в контексте сильной связанности и отказоустойчивости. Затем выясним, как его улучшить за счет изменения архитектуры на событийно-управляемую.

Предположим, что имеются два фронтенд-приложения. Можно рассматривать их как отдельные микросервисы, выполняемые на разных узлах. Они производят метрики, которые должны отправляться серверу, отвечающему за хранение значений метрик. Сценарий изображен на рис. 11.1.



Рис. 11.1. Отправка данных от двух фронтенд-приложений одному серверу

Также это означает, что у фронтенд-сервиса 1 и фронтенд-сервиса 2 существуют прямые соединения с сервисом метрик. Обмен данными происходит по стандартной схеме «запрос-ответ». Фронтенд-сервис отправляет запрос (с использованием HTTP или другого протокола), ожидает ответа и завершает работу. На практике отправка метрик может быть затратной операцией, так что на каждом из фронтенд-сервисов должен существовать пул потоков для отправки запросов.

В случае сбоя сервиса метрик оба фронтенд-сервиса не смогут отправлять данные. Это может означать, что сбой сервиса метрик каскадно распространяется на все его клиенты. Отказоустойчивость при таком решении не идеальна, так как сервис метрик становится единой точкой отказа (SPOF, single point of failure).

В реальности все бывает сложнее. Могут потребоваться N разных сервисов метрик, каждый из которых специализируется на конкретном сценарии использования. Например, может быть сервис метрик, предоставляющий информационную панель с пользовательским интерфейсом. Для автономной аналитики метрик может существовать иной сервис, обрабатывающий данные. Критические метрики могут отправляться другому сервису метрик, который специализируется на быстрой выдаче информации.

Кроме того, архитектура может расти по количеству отслеживаемых сервисов. Кроме фронтенд-сервисов могут быть базы данных, нуждающиеся в специальном контроле. Например, если бизнес занимается коммерческим обслуживанием пользователей, придется обрабатывать их платежи и счета. Все эти сервисы должны отправлять данные метрик, как показано на рис. 11.2.

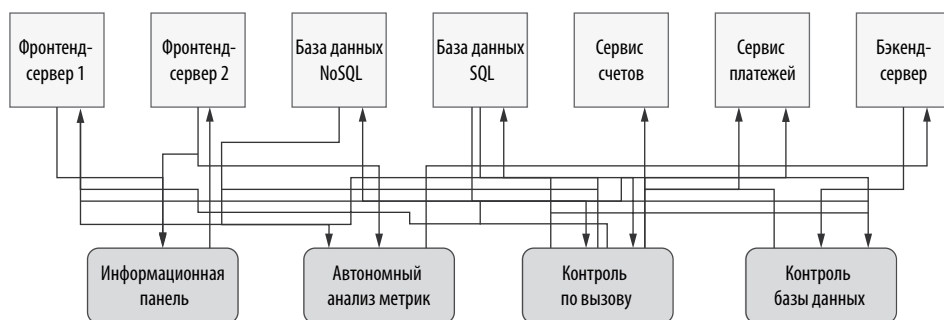


Рис. 11.2. Много сервисов — много метрик

В текущей архитектуре такая ситуация ведет к взрывному росту количества соединений между сервисами. Каждый сервис-производитель метрик должен отправлять данные N сервисам метрик. При сбое сервиса метрик во всех этих сервисах-производителях также происходит отказ. Все соединения являются прямыми, а в архитектуре образуется сильная связанность. Трудно соблюсти условия SLA, если сервис-производитель должен отправлять данные по ненадежному носителю (сети) синхронно. К счастью, событийно-управляемая архитектура решает эти проблемы, обеспечивая ослабление связей и отказоустойчивость.

В этом разделе мы представим новый компонент в улучшенной архитектуре, в которой между производителями и приложениями-потребителями добавляется новая абстрактная прослойка. Будем называть эту архитектуру системой «публикация-подписка» (*pub-sub*) или очередью событий. Очередь

событий — единственная точка интеграции между производителем и приложениями-потребителями.

Например, когда сервису нужно отправить метрику, он больше не отправляет ее напрямую сервису метрик. Допустим, необходимо отправить одну метрику на информационную панель метрик, в сервис автономной аналитики метрик и сервис контроля по вызову (рис. 11.3).

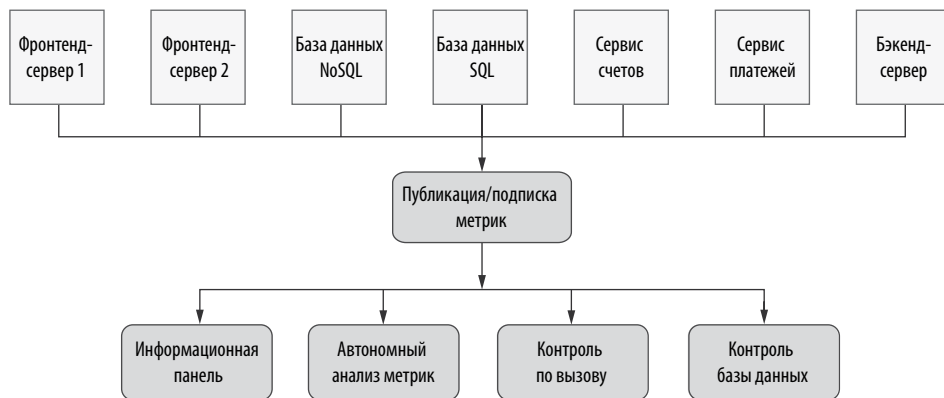


Рис. 11.3. Событийно-управляемая архитектура с абстрактной прослойкой между производителями и потребителями

В предыдущем решении фронтенд-сервис должен быть соединен с тремя разными сервисами (информационной панелью, сервисом автономной аналитики и сервисом контроля по вызову). Сбой в любом из этих сервисов будут распространяться на вызывающую службу.

Сейчас фронтенд-сервис соединен только с одним компонентом: системой pub-sub. Все приложения-потребители, которым нужна конкретная метрика, производимая фронтенд-сервисом, подписываются на события, направляемые в очередь. После того как событие окажется в очереди, все сервисы-потребители получают это событие.

Важно заметить, что наша архитектура переключилась с синхронной на асинхронную; не стало прямого соединения между производителем и потребителем. Сбой одного из сервисов метрик не влияет на фронтенд-приложение (производитель). События по-прежнему направляются в очередь, как это делается в системе pub-sub. Очередь может хранить события в течение ограниченного (или неограниченного) времени и возобновить отправку приложению-потребителю метрик, когда оно снова вернется в рабочее состояние.

Используя этот механизм, мы встраиваем отказоустойчивость в систему. Однако для его реализации необходимо хорошо понимать семантику доставки

и реализовать логику как производителя, так и потребителя. Вы узнаете, как это делается, далее в этой главе. А пока некоторые читатели заметят, что в текущей архитектуре появляется новая проблема. Компонент очереди является единственной точкой сбоя в системе. В случае сбоя система не будет работать.

Да, это так; но, к счастью, реализация проверенных систем очередей, таких как Apache Kafka или Pulsar, позволяет обеспечивать невероятно высокие условия SLA и доступность. Более того, в зависимости от конкретной ситуации можно настраивать эти системы в пользу доступности или целостности. Доступность можно улучшить, повышая количество серверов (брокеров Kafka) и/или коэффициент репликации топиков. Чем больше брокеров, тем больше сбоев перенесет система. Если данные реплицированы у N брокеров (где $N > 1$) и на одном из серверов Kafka произойдет сбой, система может оставаться доступной, так как другой брокер может начать обработку трафика в случае сбоя первого брокера.

Другое решение, которое можно реализовать для улучшения отказоустойчивости, доступности и слабого связывания системы, основано на развертывании и сопровождении N -независимых очередей. В такой конфигурации может быть одна выделенная очередь, отвечающая за метрики; другая, отвечающая за ведение журнала; и третья, для сбора контрольных событий от приложений (рис. 11.4).

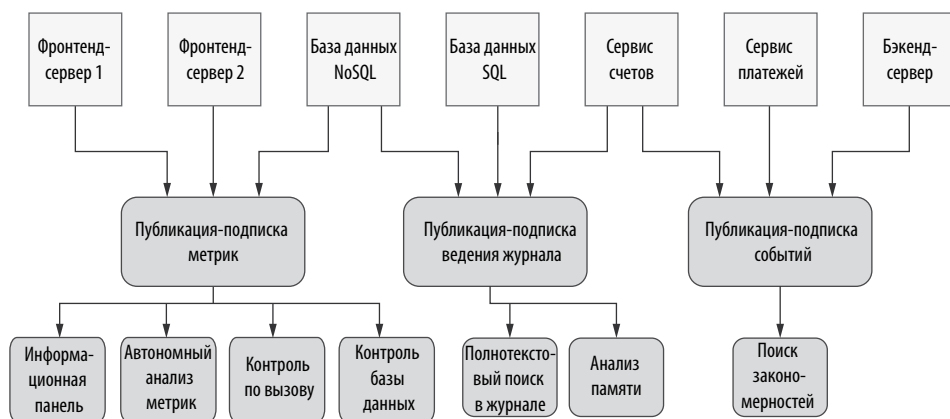


Рис. 11.4. Архитектура с несколькими независимыми системами pub-sub, разработанная для устранения единой точки отказа

В такой конфигурации уже нет единой точки отказа. Сбой одной из систем очередей не повлияет на клиентов других систем. Например, в случае отказа публикации-подписки событий вызывающие приложения все равно смогут отправлять метрики и использовать журнал, так как развернутые экземпляры публикаций-подписок могут настраиваться по-разному. Например, если сбор метрик критичен для архитектуры, можно настроить систему для повышения

доступности. Для этого можно вложить больше средств в инфраструктуру, развернуть дополнительные серверы и хранить копии данных в большем количестве мест. С другой стороны, сбор данных для отслеживания может быть не столь критичен — в таком случае недоступность некоторых данных компенсируется снижением затрат. (Вы тратите меньше денег на публикацию-подписку событий, но соглашаетесь с потерей части данных.) Разбивая функциональность очереди на N -независимых систем, можно взять все лучшее из обеих областей — слабую связанность и асинхронно отказоустойчивые системы без единой точки отказа.

В случае сбоя публикации-подписки событий вызывающее приложение может решить буферизовать (или не буферизовать) отправку событий в отдельные промежутки времени. Этот паттерн называется *Прерывателем*. Он позволит сохранить работоспособность архитектуры. Прежде чем переходить к анализу семантик доставки в событийно-управляемых архитектурах, начнем с основ Apache Kafka.

11.2. ПРОИЗВОДИТЕЛИ И ПОТРЕБИТЕЛИ НА БАЗЕ АРАСНЕ КАФКА

Прежде чем переходить к анализу гарантий доставки со стороны потребителя и производителя, стоит немного разобраться в основах архитектуры Apache Kafka. Главная конструкция, используемая на стороне как производителя, так и потребителя, — *топик* (topic). Топик представляет собой распределенную структуру данных, которая поддерживает только присоединение. Распределение достигается секционированием топика. Он может быть разбит на N разделов; чем больше разделов, тем более распределенной будет обработка. Допустим, имеется топик с именем `topicName`, состоящий из четырех разделов (рис. 11.5). Нумерация разделов начинается с 0.

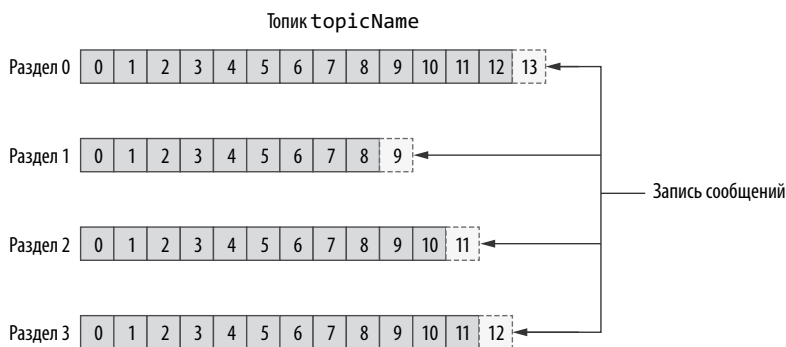


Рис. 11.5. Топик как распределенная структура, поддерживающая только присоединение

У каждого раздела имеется смещение, которое однозначно определяет одну запись в структуре, доступной только для присоединения. При отправке новой записи в топик производитель сначала вычисляет раздел, куда необходимо направить запись. Каждая запись состоит из пары «ключ-значение».

Ключ определяет секционирование для заданной записи. Например, он может содержать только значение `user_id`. При секционировании по `user_id` Kafka гарантирует, что все события одного пользователя будут отправляться одному разделу. По этой причине будет сохраняться упорядочивание событий для конкретного `user_id`. В реальных системах pub-sub топиков может быть много: в одном топике содержатся данные счетов, в другом — информация о платежах и т. д.

Когда производитель записывает сообщение, оно присоединяется в конец заданного раздела. Например, если алгоритм секционирования определяет, что событие должно быть направлено в раздел 0, оно будет присоединено к концу этой структуры. Смещение новой записи будет равно 13. Стоит отметить, что может возникнуть ситуация, в которой один раздел обрабатывает слишком много данных в случае разбалансировки разделов. Это означает, что ключ секционирования слишком узок. Можно добавить в ключ Kafka дополнительные данные, чтобы улучшить распределение по разделам.

11.2.1. Kafka на стороне потребителя

Производитель отделяется от потребителя, а потребление данных осуществляется асинхронно. Потребитель представляет собой процесс, читающий данные из топика Kafka. Как уже говорилось, топики секционируются, поэтому потребитель должен знать обо всех разделах заданного топика. У вас может быть один потребитель, который читает данные из всех разделов. Тем не менее реальные приложения структурируются в целях повышения параллелизма.

Предположим, существует топик с четырьмя разделами. Приложение должно получать данные от этих четырех разделов. В такой конфигурации в зависимости от требований к производительности могут быть развернуты до четырех потребителей. Каждый из них отвечает за чтение событий от одного раздела. Если их больше, чем разделов, дополнительные потребители будут простаивать, поскольку разделы всех топиков уже назначены потребителям.

Представим чуть более сложную ситуацию. Топик все еще состоит из четырех разделов, но иметь четыре приложения-потребителя не обязательно (рис. 11.6). Тесты производительности показывают, что трех процессов-потребителей достаточно для обеспечения необходимой пропускной способности.

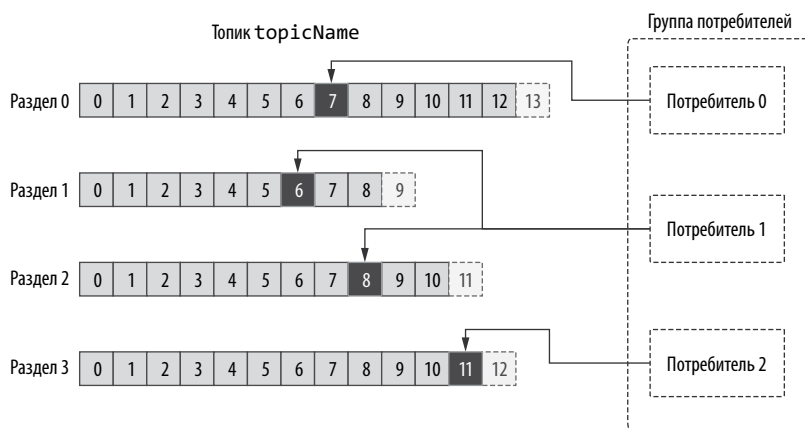


Рис. 11.6. Назначение топиков в группе с несколькими потребителями

Это нормально. В такой конфигурации один из процессов-потребителей (потребитель 1) получает два раздела. С другой стороны, потребитель 0 и потребитель 2 обрабатывают один раздел. Потребитель 1 обрабатывает два раздела, 1 и 2. Отметим, что в реальном сценарии распределение разделов по потребителям может быть другим. Тем не менее в рассматриваемом сценарии каждому потребителю будет назначен как минимум один раздел.

Важно, что обработка дополнительных разделов не может распределяться между *N* узлами, так как это нарушит гарантии упорядочения внутри раздела. Многие потребители будут получать события с одинаковыми ключами разделов, и в такой конфигурации обеспечить упорядочение не удастся. Следовательно, в Apache Kafka такая ситуация невозможна.

Описанное решение (четыре раздела и три потребителя) создает проблемы, потому что один из потребителей обрабатывает вдвое больше событий, чем другие. По этой причине в реальной ситуации нужно выбирать четное количество потребителей. При четырех разделах создание двух потребителей приведет к тому, что два потребителя будут обрабатывать одинаковый объем трафика. Если потребуется более высокая пропускная способность, можно использовать четыре потребителя.

ПРИМЕЧАНИЕ

Важно заранее выбрать количество разделов при создании топика. Будьте внимательны и выбирайте число на основании тестов производительности и эмпирических данных.

Предположим, стало понятно, что выбранного количества разделов недостаточно для обработки трафика. В таком случае необходимо создать новый топик с большим количеством разделов и перенести старый топик в новый. Однако это требует больших затрат ресурсов и времени.

Одно из важнейших преимуществ использования Apache Kafka — способность развертывания N -независимых приложений-потребителей. Каждое приложение в Kafka называется *группой потребителей*. Группа содержит N потребителей. Это позволяет реализовать сценарий, описанный в разделе о публикации-подписке.

Один топик может потребляться несколькими приложениями. Каждое приложение потребляет данные от одного топика независимо и в собственном темпе. Например, если у приложения информационной панели метрик (со специализированной группой `customer_group`) нет высоких требований к пропускной способности, оно может выполняться на одном физическом узле с одним процессом-потребителем Kafka. С другой стороны, приложение контроля по вызову может быть более критичным и чувствительным к производительности. Группа потребителей может иметь N потребителей, которые будут быстрее обрабатывать данные.

11.2.2. Конфигурация брокеров Kafka

Наконец, проанализируем полную конфигурацию Apache Kafka с внедрением по N брокерам. Возьмем простейший сценарий, при котором Kafka разворачивается на двух физических машинах. На каждой из этих машин существует один брокер Kafka. Проанализируем топик T с двумя разделами. Это означает, что максимальный параллелизм на сторонах производителя и потребителя равен 2 (количество разделов). Помимо этого коэффициент репликации топика T задан равным 2, так что каждое событие будет (в конечном итоге) сохранено на обоих брокерах.

Предполагается, что в описанной конфигурации есть только один производитель и только один потребитель. На практике можно иметь до двух производителей и до удвоенного количества потребителей на каждую группу. Тем не менее конфигурация с одним производителем и одним потребителем упрощает описание конфигурации брокеров Kafka. На рис. 11.7 изображена конфигурация для нашего сценария использования.

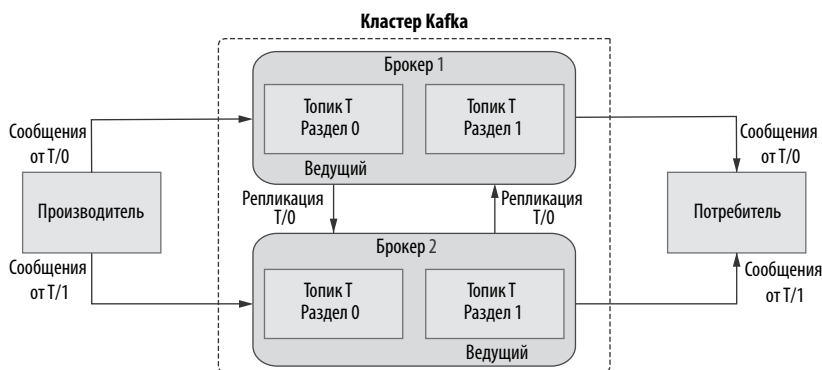


Рис. 11.7. Конфигурация с несколькими брокерами Kafka

Топик Т состоит из двух разделов. Каждый из них реплицируется на обоих брокерах, поскольку коэффициент репликации этого топика равен 2. Если задать ему значение 1, каждый раздел будет храниться только на одном брокере. Раздел работает по модели «ведущий-ведомый»; только один брокер является ведущим для заданного раздела. В нашей схеме брокер 1 является ведущим для топика Т, раздел 0, а брокер 2 — для топика Т, раздел 1.

Помните: чем выше коэффициент репликации, тем больше ресурсов требуется кластеру. Например, с коэффициентом репликации 2 потребуется вдвое больше дискового пространства, чем с коэффициентом репликации 1. Это связано с тем, что данные должны храниться в двух физических местах. С коэффициентом репликации 3 объем дискового пространства утраивается. Кроме того, сохранение данных на большем количестве брокеров требует большего сетевого трафика и вычислительной мощности процессора. Данные, задействованные в репликации, должны передаваться большему количеству брокеров по сети.

Когда производитель отправляет данные в раздел темы, они направляются ведущему для этого раздела. Затем данные реплицируются у ведомого брокера, хранящего данные на случай сбоя. Если брокер 1 перестанет работать, брокер 2 станет ведущим для этого раздела. Процесс-потребитель должен вести список ведущих для разделов всех топиков. Это позволяет ему потреблять данные от соответствующего раздела. В случае сбоя брокера процесс перебалансировки обновляет информацию ведущих для всех разделов. Итак, теперь вы понимаете, как работает Kafka, и мы можем перейти к анализу семантики доставки производителя.

11.3. ЛОГИКА ПРОИЗВОДИТЕЛЯ

Начнем с логики производителей в Kafka. Производитель Apache Kafka (<http://mng.bz/lad2>) станет нашей основной точкой входа для отправки данных топикю Kafka. Производитель Kafka можно настраивать с различными параметрами (<http://mng.bz/BxP1>). Тем не менее есть три обязательных настройки, которые задать необходимо.

Первая — список брокеров Kafka `bootstrap.servers`. В нем должны быть перечислены брокеры Kafka в кластере. Список используется производителем для определения того, куда должны отправляться события. Кроме того, каждая запись Kafka содержит пару «ключ-значение». Для обоих компонентов необходимо задать сериализаторы, предоставляющие логику преобразования объекта Java (например, `String`) в массив байтов, передаваемый топикю Kafka. В следующем

листинге приведен пример конфигурации производителей Kafka, использующей библиотеку Spring Kafka (<http://mng.bz/dojo>).

Листинг 11.1. Создание конфигурации производителя Kafka

```
@Configuration
public class SenderConfig {

    @Value("${kafka.bootstrap-servers}")
    private String bootstrapServers;

    @Bean
    public Map<String, Object> producerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            IntegerSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        return props;
    }

    @Bean
    public ProducerFactory<Integer, String> producerFactory() {
        return new DefaultKafkaProducerFactory<>(producerConfigs());
    }

    @Bean
    public Producer<Integer, String> producer() {
        return producerFactory().createProducer();
    }

    @Bean
    public KafkaTemplate<Integer, String> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }

    @Bean
    public Sender sender() {
        return new Sender();
    }
}
```

Карта настроек
производителя Kafka

Producer обрабатывает
целый ключ и строковое
значение

Sender — абстракция Spring
на базе низкоуровневого
производителя Kafka

Логика производителя использует ранее созданный экземпляр **Producer** для отправки данных в топик Kafka. Так как логика является асинхронной, действие **Producer** выполняется без блокирования и возвращает **Future**. Отметим, что совместное использование одного экземпляра **Producer** в нескольких потоках для отправки данных по нескольким топикам безопасно. **Producer** получает в аргументах топик, ключ раздела и фактическое значение. На основании ключа раздела он направляет запрос в раздел соответствующего топика, как показано в следующем листинге.

Листинг 11.2. Создание производителя Kafka

```

@Autowired private Producer<Integer, String> producer;

public Future<RecordMetadata> sendAsync
    (String topic, String data, Integer partitionKey) {
    LOGGER.info("sending data='{ }' to topic='{ }'", data, topic);
    try {
        return producer.send(
            new ProducerRecord<>(topic, partitionKey, data),
            (recordMetadata, e) -> {
                if (e != null) {
                    LOGGER.error("error while sending data:" + data, e);
                }
            });
    } finally {
        producer.flush();
    }
}

```

Возвращает Future
 Producer получает топик, ключ раздела и отправляемые данные
 Передает ProducerRecord непосредственно производителю Kafka
 Выполняет асинхронный обратный вызов после успешной отправки записи

Операция отправки выполняется асинхронно, и можно зарегистрировать обратный вызов, который будет выполнен после завершения отправки. Обратный вызов проверяет, что исключение не равно `null`. Если оно отлично от `null`, значит, при отправке произошел сбой.

За простой логикой `send()` скрывается значительная сложность. Еще раз вкратце повторим эту логику, проанализировав диаграмму на рис. 11.8.

Сначала создается экземпляр `ProducerRecord`. Он содержит топик, ключ и значение. Можно предоставить раздел или вычислить его по значению ключа. Если ключ не предоставляется (он равен `null`), сообщения распределяются по циклическому алгоритму. Затем данные сериализуются в массив байтов. Наконец, блок секционирования определяет раздел, в который необходимо отправить запись.

Важно заметить, что записи группируются на стороне производителя на уровне раздела темы. Это означает, что один пакет может содержать N записей для того же раздела. Когда отправка завершится успешно, возвращаются метаданные для всех отправленных записей. В частности, в них содержится смещение для раздела, которому отсылаются данные. Если происходит сбой, попытка отправки повторяется. Повторными попытками управляет параметр `retries` (<http://mng.bz/VIXy>). Если попытки еще не исчерпаны, выполняется повторная попытка отправки пакета записей. Если попыток не осталось, на сторону вызова распространяется исключение.

Стоит заметить, что повторная попытка отправки пакета для заданного раздела может нарушить гарантии упорядочения внутри раздела. Если первый запрос завершается неудачей и планируется повторная попытка, второй запрос может выполняться успешно до запланированной повторной попытки. В таком случае обработка пакетов чередуется (происходит примерно то же, что и в предыдущей главе). Это приводит к нарушению порядка событий внутри раздела.

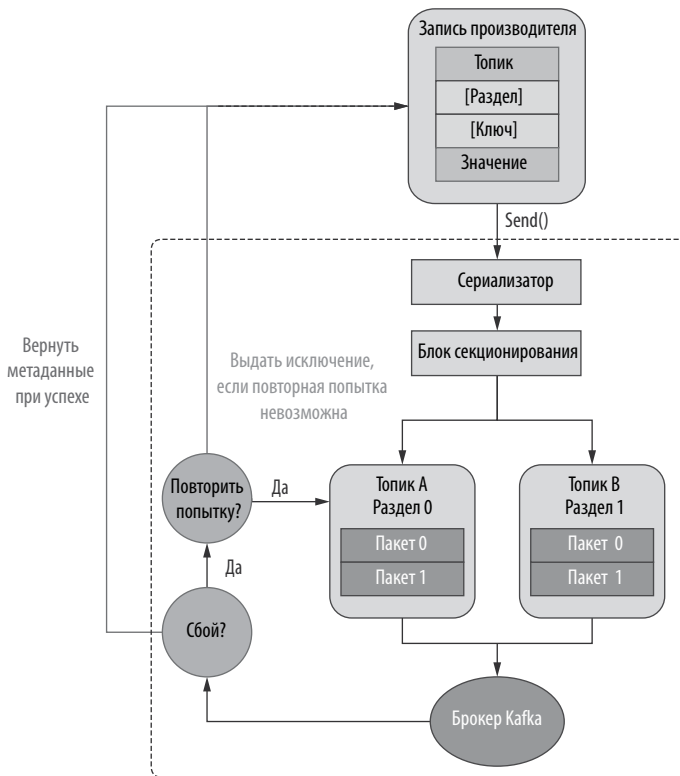


Рис. 11.8. Логика производителя Kafka для операции отправки

Если повторные попытки включены (а они включены по умолчанию), возникает вероятность дублирования. Производитель, работающий в этом режиме, предоставляет гарантию доставки «не менее одного» — одна запись может быть отправлена несколько раз. Повторные попытки чрезвычайно важны для обеспечения отказоустойчивости и стабильности логики производителя.

11.3.1. Выбор между целостностью данных и доступностью для производителя

Другой важный компромисс, который необходимо принять на стороне производителя, — выбор между целостностью данных и доступностью. Допустим, имеется кластер с двумя брокерами и топик A реплицируется на обоих брокерах. Для простоты будем считать, что в топике A только один раздел (поведение будет одинаковым для N разделов). При отправке производителем данных в топик появляются три варианта, относящихся к количеству подтвержденных ответов брокера (<http://mng.bz/xvQd>). Во всех этих вариантах характеристики целостности

и доступности будут разными. Для начала рассмотрим вариант, в котором параметру `acks` присвоено значение `all`, как показано на рис. 11.9.



Рис. 11.9. В режиме `acks=all` предпочтение отдается целостности (производитель ожидает подтверждения от сервера)

Если топик создается с коэффициентом репликации 2, запись должна быть успешно сохранена и подтверждена всеми брокерами (двумя в нашем сценарии использования). Для нашего топика А и раздела 0 ведущим является брокер 1. Производитель отправляет данные этому брокеру. Так как параметру `acks` присвоено значение `all`, ведущий распространяет запись брокеру 2 (ведомому). Когда данные будут успешно сохранены на ведомом, ведущий получает ответ из возврата признака успеха производителю.

Если на одном брокере происходит сбой, то данные остаются целостными. Это гарантирует, что на обоих брокерах для топика 1 будут одни и те же данные. Но в данном случае (сбой у одного брокера) невозможно обеспечить коэффициент репликации, которому присвоено значение 2. Когда это происходит, система становится недоступной. Таким образом, мы пожертвовали доступностью всей системы ради целостности данных.

В реальной конфигурации брокеров должно быть больше. Если бы у нас было три брокера, а коэффициенту репликации было присвоено значение 2, то сбой одного брокера не привел бы к недоступности системы. Пока два брокера находятся в рабочем состоянии, можно успешно отправлять данные.

Выбор количества брокеров и коэффициента репликации топиков зависит от сценария использования. Чтобы определить его, сначала определите максимальное количество запросов в секунду (и в мегабайтах в секунду), которые должен обрабатывать кластер. Зная это число, можно протестировать производительность одного брокера и найти его максимальную пропускную способность. Вы можете

воспользоваться сетевыми ресурсами (например, <http://mng.bz/Axwo>), но учтите, что максимальная пропускная способность зависит от типа используемого вами компьютера. Производительность диска, количество процессоров, объем памяти — все это влияет на пропускную способность.

Когда вы будете знать максимальную пропускную способность на одного брокера, можно вычислить количество брокеров, необходимых для обработки трафика. Но для обеспечения высокой доступности и целостности в системе Kafka следует повысить коэффициент репликации темы. Он зависит от индивидуальных потребностей, и выбирать его следует тщательно. Чем выше коэффициент репликации топика, тем большую пропускную способность должен обеспечивать кластер. Например, при выборе коэффициента репликации 2 сетевой трафик удвоится. Так как для хранения данных нужны два брокера Kafka, понадобится вдвое больше места на диске.

ПРИМЕЧАНИЕ

Создание кластера Kafka, готового к реальной эксплуатации, — непростая задача, поэтому я рекомендую поэкспериментировать и изучить литературу, чтобы найти оптимальную конфигурацию.

Рассмотрим ситуацию, в которой параметру `acks` присвоено значение 1. В таком случае производитель ожидает подтверждения сохранения данных только от одного брокера (ведущего). Эта конфигурация представлена на рис. 11.10.

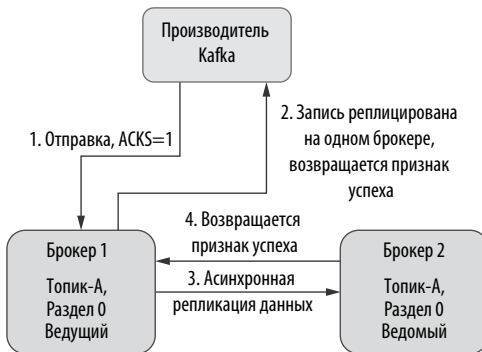


Рис. 11.10. При `acks=1` предпочтение отдается доступности данных

В этом сценарии данные по-прежнему реплицированы на количестве брокеров, равном коэффициенту репликации топика А. Тем не менее процесс репликации происходит асинхронно. Когда ведущий успешно сохранит запись производителя, он немедленно вернет признак успеха на сторону вызова.

В фоновом режиме происходит синхронизация данных с ведомым. Однако существует вероятность, что сбой произойдет до того, как ведомый сохранит данные.

В таком сценарии данные не будут реплицированы на втором брокере. Так как производитель ожидает только одного подтверждения, он не знает о сбое в ходе фоновой операции. Если на брокере 1 произойдет сбой, а у брокера 2 еще нет всех актуальных данных топика А, возникает риск потери данных. С другой стороны, даже если в кластере функционирует только один брокер, производитель все еще отправляет данные в топик А. В этом сценарии мы жертвуем целостностью ради доступности топика А.

Существует и третье возможное значение `acks` — параметру можно присвоить 0. В этом случае производитель не ожидает никаких подтверждений от брокера Kafka. Операция выполняется по принципу «выстрелил и забыл» и ограниченно применяется в реальных условиях. Существует высокая вероятность, что данные будут утеряны, причем абсолютно незаметно.

Теперь вы понимаете, что происходит на стороне производителя, и можно перейти к стороне потребителя Kafka. Мы реализуем логику потребителя с разными вариантами семантики доставки.

11.4. КОД ПОТРЕБИТЕЛЯ И РАЗНЫЕ СЕМАНТИКИ ДОСТАВКИ

После того как данные будут успешно сохранены в структуре топика, доступной только для присоединения, код потребителя Kafka сможет загрузить его. Так как для топиков можно настроить время удержания, события с самыми старыми смещениями удаляются по истечении этого времени. Время удержания может быть бесконечным; в этом случае старые события вообще не будут удаляться. Начнем с примера кода потребителя.

При настройке потребителя также необходимо передать список брокеров Kafka. Как вы, возможно, помните, компонент производителя должен использовать сериализатор для преобразования объекта в массив байтов. Потребитель должен провести обратное преобразование: из байтов в объекты. Следовательно, необходимо предоставить классы десериализаторов пар «ключ-значение». Каждый потребитель работает внутри группы, поэтому также необходимо передать идентификатор группы, который будет использоваться потребителем.

Важно понимать, что смещения топика Kafka отслеживаются конкретной группой потребителей. Это означает, что при чтении потребителем пакетов событий из топика смещение должно быть зафиксирован, указывая на корректную обработку событий. В случае сбоя другой потребитель из группы может возобновить обработку от последнего зафиксированного смещения.

Способ фиксации и возобновления обработки влияет на семантику доставки, предоставляемую приложением-потребителем. Начнем с простейшего случая,

в котором потребитель Kafka автоматически фиксирует смещения за нас. Того же эффекта можно добиться, присвоив `enable.auto.commit` (<http://mng.bz/ZzgR>) значение `true`, как показано в следующем листинге.

Листинг 11.3. Настройка потребителя Kafka

```
@Bean
public Map<String, Object> consumerConfigs() {
    Map<String, Object> props = new HashMap<>();

    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
        IntegerDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        StringDeserializer.class);
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "receiver");
    props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
    return props;
}
```

Эта конфигурация может использоваться для создания потребителя Kafka. Потребитель может работать для N топиков и может совместно использоваться между потоками. Только не забудьте подписаться на топик, информация которого должна потребляться, как показано в следующем листинге.

Листинг 11.4. Создание потребителя Kafka с автоматической фиксацией

```
public KafkaConsumerAutoCommit(Map<String, Object> properties, String topic) {
    consumer = new KafkaConsumer<>(properties);
    consumer.subscribe(Collections.singletonList(topic));
}

public void startConsuming() {
    try {
        while (true) {
            ConsumerRecords<Integer, String> records =
                consumer.poll(Duration.ofMillis(100));
            for (ConsumerRecord<Integer, String> record : records) {
                LOGGER.debug(
                    "topic = {}, partition = {}, offset = {}, key = {}, value = {}",
                    record.topic(),
                    record.partition(),
                    record.offset(),
                    record.key(),
                    record.value());
                logicProcessing(record);
            }
        }
    } finally {
        consumer.close();
    }
}
```

Потребитель получает события от топика, на который он подписан

Обработка выполняется в цикле while

Опрашивает все доступные записи с ожиданием не более 100 мс

Возвращает пакет, который может содержать записи для всех топиков с подпиской

Метод `startConsuming()` в цикле вызывает метод `poll()` потребителя и ожидает результата в течение 100 мс. Этот метод возвращает пакет записей, который должен быть обработан. Каждая запись содержит ключи и значения, а также информацию отслеживания — например, топик и раздел. Метод `offset()` возвращает точное смещение конкретной записи в разделе заданного топика. Наконец, мы перебираем пакет записей и обрабатываем каждую из них.

Когда потребитель работает в режиме автофиксации, он закрепляет смещения в фоновом режиме каждые N мс в соответствии с настройкой `auto.commit.interval.ms` (<http://mng.bz/REnZ>), которая по умолчанию равна 5 с.

Представьте, что ваше приложение обрабатывает 100 событий в секунду, как показано на рис. 11.11. Допустим, они поступили пятью пакетами. В таком сценарии смещение фиксируется после обработки 500 событий. Если сбой в приложении происходит менее чем за 5 с, смещение не фиксируется. Тогда последнее известное смещение для этой обработки равно 0.

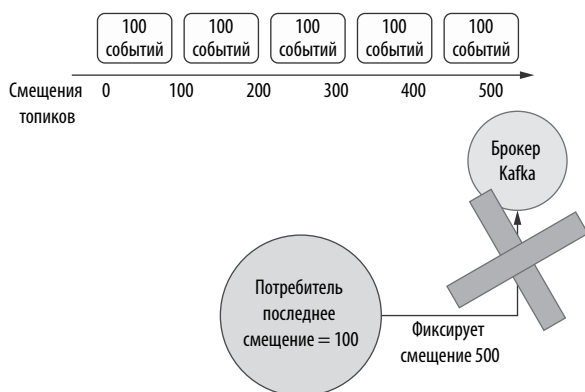


Рис. 11.11. Потребительская автофиксация для процесса с 100 событиями в секунду

Если другой потребитель в этой группе возобновляет обработку из-за сбоя, он видит, что последнее фиксированное смещение равно 0. Он опрашивает 500 событий, которые, возможно, уже были обработаны потребителем с предшествующим сбоем, а это означает вероятность обработки 500 дублированных событий. Это пример семантики доставки «не менее одного». Наш потребитель может получить событие один раз в случае успешной фиксации. Но если фиксация не будет успешной, то другой потребитель снова обработает данные.

11.4.1. Ручная фиксация у потребителя

Положение дел можно поправить за счет использования ручной фиксации. Сначала необходимо отключить автофиксацию, присвоив параметру значение `false`, как показано в следующем листинге:

Листинг 11.5. Отключение автофиксации

```
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
```

После этого потребитель уже не фиксирует смещения автоматически. Эта обязанность теперь возлагается на вас. Самое важное решение, которое необходимо принять, — фиксировать смещения при входе в систему или после обработки. Если вы хотите поддерживать семантику доставки «не менее одного», смещение должно фиксироваться после логики обработки. Тем самым гарантируется, что сообщение его успешной обработки будет помечено как зафиксированное. Процесс показан в следующем листинге.

Листинг 11.6. Синхронная фиксация

```
public void startConsuming() {
    try {
        while (true) {
            ConsumerRecords<Integer, String> records =
consumer.poll(Duration.ofMillis(100));
            for (ConsumerRecord<Integer, String> record : records) {
                logicProcessing(record);
                try {
                    consumer.commitSync();
                } catch (CommitFailedException e) {
                    LOGGER.error("commit failed", e);
                }
            }
        }
    } finally {
        consumer.close();
    }
}
```

Этот код потребления отличается только ручной фиксацией

В листинге 11.6 для достижения цели используется метод `commit()`. Он фиксирует смещения всех разделов, назначенных этому конкретному потребителю. Важно заметить, что метод `commit()` является блокирующим. Это означает, что обработка продолжится только после фиксации смещений. Хотя такой подход обеспечивает безопасность, он может повлиять на общую производительность нового решения, так что операция `commit()` может оказаться весьма затратной.

Если эти затраты неприемлемы, можно воспользоваться методом `commitAsync()`, который не блокирует обрабатывающий поток. Однако при асинхронной фиксации необходимо внимательно следить за обработкой ошибок, потому что исключения не распространяются в основной поток вызывающей стороны. Реализация с `commitAsync()` приведена в следующем листинге.

Листинг 11.7. Асинхронная фиксация

```
consumer.commitAsync(
    (offsets, exception) -> {
        if (exception != null) LOGGER.error(
➡ "Commit failed for offsets {}", offsets, exception);
    });
```

Иногда асинхронная фиксация завершается ошибкой, но фиксация для последующего пакета событий проходит успешно. В таком сценарии это никак не повлияет на работу системы, потому что правильное смещение, зафиксированное последующим действием, было сохранено.

Рассмотрим ситуацию, в которой требуется зафиксировать смещения перед тем, как логика обработает события. В таком случае сбой в логике обработки пройдет незамеченным для брокера Kafka. Смещение уже закреплено, поэтому, когда логика потребителя возобновит обработку, предыдущий пакет не будет обработан заново.

Если метод `logicProcessing()` не завершится успешно, некоторые события не будут обработаны. В этом случае возникает риск потери событий. В такой системе действуют гарантии доставки «не более одного». Одно событие будет обработано только один раз (но существует вероятность, что оно не будет обработано ни разу).

11.4.2. Перезапуск от самых ранних или поздних смещений

Существует и второй аспект, влияющий на гарантии доставки приложений-потребителей. Рассмотрим сценарий с топиком из 10 записей (а следовательно, с 10 смещениями). Клиентское приложение получает все записи из пакета. Пакет может содержать от 1 до 10 событий и фиксировать смещение, равное N (любое число от 0 до 10, равное количеству событий в пакете). К сожалению, на этапе фиксации в приложении происходит сбой. В этом случае мы не знаем, сколько событий обработало приложение-потребитель. На это может влиять много факторов, включая тайм-аут пула потребителей, размер пакета и т. д. Когда приложение перезапускается, возможны два варианта возобновления обработки.

В таком сценарии обеими стратегиями возобновления обработки управляет параметр `auto.offset.reset` (<http://mng.bz/2jqg>). Если ему присваивается значение `earliest`, возобновление обработки событий начнется с последнего зафиксированного смещения для разделов топика (если оно есть). Если смещения нет, повторная обработка всех событий начинается заново. Эта стратегия проиллюстрирована на рис. 11.12.

В этой ситуации у приложения-потребителя могут появиться дубликаты. Дело в том, что сбой логики потребителя могут произойти в любой момент во время обработки последующих записей. При одном перезапуске можно получить до 20 дубликатов (2×10 событий). Такая стратегия сброса смещений обеспечивает семантику доставки «не менее одного».

0	1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	---	----

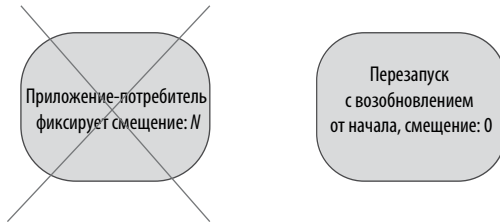


Рис. 11.12. Перезапуск с более раннего смещения

За этой стратегией можно понаблюдать в интеграционном тесте. В этом тесте мы передадим потребителю Kafka значение `OffsetResetStrategy.EARLIEST`, как показано в следующем листинге.

Листинг 11.8. Тестирование стратегии возврата к самому раннему смещению

```
// Дано
ExecutorService executorService = Executors.newSingleThreadExecutor();
String message = "Send unique message " + UUID.randomUUID().toString();

KafkaConsumerWrapper kafkaConsumer =
    new KafkaConsumerWrapperCommitOffsetsOnRebalancing(
        KafkaTestUtils.consumerProps(
            "group_id" + UUID.randomUUID().toString(),
            "false",
            AllSpringKafkaTests.embeddedKafka),
        CONSUMER_TEST_TOPIC,
        OffsetResetStrategy.EARLIEST);

// Если
sendTenMessages(message);
executorService.submit(kafkaConsumer::startConsuming);
sendTenMessages(message);

// To
executorService.awaitTermination(4, TimeUnit.SECONDS);
executorService.shutdown();
assertThat(kafkaConsumer.getConsumedEvents()
    ➔ .size()).isGreaterThanOrEqualTo(20);
```

← Передает стратегию самого раннего смещения

← Вызывает `startConsuming()` после отправки 10 записей

← Получает все 20 записей

После отправки 10 событий, запуска потребителя и отправки еще 10 событий можно проверить количество полученных записей. Здесь потребитель получает все 20 событий, опубликованные производителем.

Также можно выбрать стратегию последнего смещения. С этой стратегией возобновление обработки после сбоя начинается с последнего смещения для этого

топика. В нашем сценарии приложение начинает со смещения 10 или выше. Данный сценарий актуален, если производитель добавляет новые события. Эта стратегия показана на рис. 11.13.

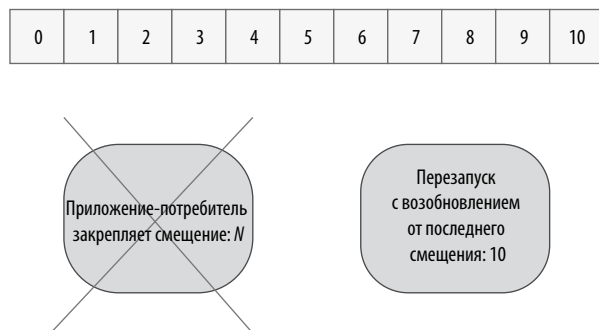


Рис. 11.13. Перезапуск с последнего смещения

В этом сценарии приложение может потерять некоторые события, доставленные до сбоя. Возможно, они были доставлены, но не обработаны. В нем также не будет дубликатов, но возможна потеря событий. Использование стратегии последнего смещения предоставляет гарантию доставки «не более одного».

Логика тестирования похожа на ту, что была использована в предыдущем примере. Сначала мы создаем потребителя Kafka со стратегией `OffsetResetStrategy.LATEST`, как показано в листинге 11.9. Передавать этот параметр не обязательно, потому что это значение используется в Kafka по умолчанию; здесь оно передается для полной ясности и выражения намерений. Потребитель создается для случайного идентификатора группы (чтобы начать с несуществующего смещения), и смещения не фиксируются автоматически. Затем мы отправляем 10 сообщений в топик Kafka. После того как сообщения будут отправлены, можно запустить потребителя Kafka. Когда он будет запущен, передаются следующие 10 сообщений.

Листинг 11.9. Тестирование стратегии возврата к последнему смещению

```
// Дано
ExecutorService executorService = Executors.newSingleThreadExecutor();
String message = "Send unique message " + UUID.randomUUID().toString();

KafkaConsumerWrapper kafkaConsumer =
    new KafkaConsumerWrapperCommitOffsetsOnRebalancing(
        KafkaTestUtils.consumerProps(
            "group_id" + UUID.randomUUID().toString(), ←
            "false",
            AllSpringKafkaTests.embeddedKafka(),
            CONSUMER_TEST_TOPIC,
            OffsetResetStrategy.LATEST); ←
```

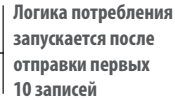
Генерирует группу потребителей динамически, чтобы избежать возможных конфликтов с другими тестами потребителей

Передаёт стратегию последнего смещения

```
// Если
sendTenMessages(message);

executorService.submit(kafkaConsumer::startConsuming);
sendTenMessages(message);

// To
executorService.awaitTermination(4, TimeUnit.SECONDS);
executorService.shutdown();
assertThat(kafkaConsumer.getConsumedEvents().size()).isLessThanOrEqualTo(10);
```



Логика потребления
запускается после
отправки первых
10 записей

Возможно, вы заметили, что потребитель Kafka получил 10 событий. События, опубликованные до запуска потребителя, не учитываются этим потребителем.

Этот интеграционный тест похож на предыдущий. У обеих ситуаций есть свои плюсы и минусы и свои сценарии применения. Если в вашей области задержки критичны и необходимо реагировать на последние события, вариант с возобновлением с последнего смещения может подойти. Например, для системы оповещения могут не представлять интереса события, доставленные минуты назад. Реагирование на устаревшие данные не принесет пользы. С другой стороны, если ваша система критична к правильности информации, следует обрабатывать все события и защищаться от дубликатов. Например, если в платежной системе произойдет сбой, необходимо возобновить обработку с точки сбоя и обработать все незавершенные платежи.

11.4.3. Семантика «фактически ровно один»

Построить систему, предоставляющую гарантию «ровно один», сложно. До сих пор мы представили две возможные семантики доставки: «не менее одного» и «не более одного». Если логика системы неидемпотентна и никакие события не должны теряться, необходима разновидность семантики «ровно один».

На практике системы, предоставляющие семантику «фактически ровно один», часто строятся на базе семантики доставки «не менее одного». Вы узнали в предыдущей главе, что реализация логики дедупликации может предоставить разновидность семантики «фактически ровно один». Мы говорим *фактически*, потому что на каком-то уровне события могут дублироваться. Например, они могут дублироваться логикой повторных попыток на стороне производителя. В этом случае такие дубликаты скрываются от системы, которая ожидает ровно однократной доставки.

Apache Kafka формирует семантику «фактически ровно один», реализуя разновидность распределенных транзакций. В архитектуре Kafka дубликаты могут существовать на уровнях как производителя, так и потребителя. По умолчанию производитель также может получать дубликаты при перезапуске из-за поведения закрепления смещений, описанного в разделе 11.4.

Для решения этой проблемы Apache Kafka применяет транзакции. Они начинаются на стороне производителя до отправки нового события в топик Kafka. При этом идентификатор транзакции `transactional_id` (<http://mng.bz/1jPX>) используется для обеспечения семантики «фактически ровно один» в пределах транзакции. Каждой записи назначается идентификатор транзакции. В случае сбоя при отправке операция отменяется и Kafka гарантирует, что заданная запись не будет присутствовать в топике Kafka. Можно сделать повторную попытку с другой транзакцией. Тем не менее транзакция только охватывает логику внутри заданного производителя Kafka. Если логика производителя в сервисе базируется на внешнем событии (получаемом от другого кластера Kafka или из HTTP), все равно можно получить дубликат.

Событие, инициирующее отправку производителем, может доставляться с семантикой гарантий «не менее одного» (рис. 11.14). Если система, использующая транзакции Kafka, не защищается от таких дубликатов, эти события рассматриваются производителем как два независимых события.

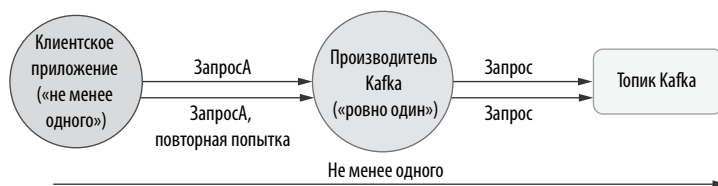


Рис. 11.14. Семантика «ровно один» в контексте «не менее одного»

Предположим, что клиентское приложение не реализует транзакции и предоставляет семантику доставки «не менее одного». В случае сбоя оно может повторить попытку запроса. Приложение, логика которого основана на этом клиентском событии, использует транзакции производителя Kafka для предоставления семантики «фактически ровно один». Важно заметить, что при этом отсутствует защита от дубликатов.

С точки зрения приложения-производителя Kafka запросы различны. Если механизм дедупликации не реализован, невозможно определить, являются ли запросы дубликатами. Затем запросы доставляются с использованием транзакций, и оба — с гарантиями «фактически ровно один». Тем не менее с позиций логики системы одно событие было отправлено Kafka дважды (оба события являются дубликатами). Следовательно, на логическом уровне действует гарантия доставки «не менее одного».

Очевидно, семантика «фактически ровно один» может работать, но только в том случае, если она поддерживается всеми компонентами бизнес-процесса приложения. На практике она может содержать N стадий обработки, взаимодействия и обмена данными по системе pub-sub, протоколу HTTP или другому механизму. Это будет означать, что весь конвейер должен быть заключен в одну транзакцию.

Такое решение может оказаться непрочным и не отказоустойчивым. В случае сбоя на какой-либо из стадий может оказаться, что бизнес-процесс не сможет продолжать выполнение без ручного вмешательства оператора для исправления нарушенной транзакции.

Если вы хотите использовать в приложении семантику «фактически ровно один», будьте внимательны с его производительностью и доступностью. Решение об использовании этого механизма необходимо принимать на основе результатов тщательного тестирования производительности и хаотического тестирования продукта. В следующем разделе мы покажем, как применить семантику доставки Kafka для улучшения отказоустойчивости системы.

11.5. ИСПОЛЬЗОВАНИЕ ГАРАНТИЙ ДОСТАВКИ ДЛЯ ОБЕСПЕЧЕНИЯ ОТКАЗОУСТОЙЧИВОСТИ

Рассмотрим сценарий с двумя системами, работающими на базе обработки событий; единственной точкой интеграции между этими системами является топик Kafka. Предположим, что сервис оформления заказов генерирует событие платежа с использованием асинхронной логики производителя Kafka. Затем биллинговый сервис потребляет данные из топика и обрабатывает их, как показано на рис. 11.15.



Рис. 11.15. Сервисы биллинга и оформления заказов, работающие как событийные процессы

Предположим, что сервис оформления заказов в среднем отправляет 50 запросов в секунду. Условия SLA сервиса биллинга гарантируют обработку до 100 запросов в секунду (рис. 11.16). Теперь рассмотрим ситуацию, в которой биллинговый сервис сбоят или останавливается в ходе развертывания новой версии. В это время биллинговый сервис не может потреблять события из топика Kafka. Ослабление связанности между этими двумя сервисами обеспечивает отказоустойчивость на уровне сервиса оформления заказов.

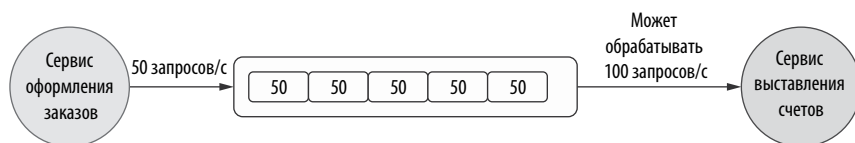


Рис. 11.16. Буферизация событий в топике Kafka

События платежей будут создаваться, даже если биллинговый сервис не работает. Однако они будут сохраняться в топике Kafka. Допустим, сервис биллинга перезапускается и начинает нормально работать через 5 секунд. За это время сервис оформления заказов отправляет в топик Kafka 250 событий (50 запросов в секунду \times 5 с). Эти события буферизуются и сохраняются в топике Kafka. Приложение-потребитель сможет возобновить обработку буферизованных событий, когда его работоспособность восстановится.

Важно, что возобновление обработки от последнего обработанного события может быть реализовано, если биллинговый сервис работает с гарантиями доставки «не менее одного». Это означает, что потребитель должен фиксировать смещения должным образом (после успешно обработанной логики) и для стратегии сброса смещения должно быть выбрано значение `earliest`. 250 событий, буферизованных в топике, должны потребляться наряду с обычным трафиком от сервиса оформления заказов; в противном случае биллинговый сервис будет отставать от входящего трафика.

Как мы знаем, биллинговый сервис способен обрабатывать 100 запросов в секунду. Помимо этого, он должен потреблять 50 событий в секунду в ходе обычной работы. В таком сценарии сервису биллинга понадобятся дополнительные 2,5 секунды для обработки буферизованных событий. В это время может наблюдаться задержка обработки 2,5 с и более. Дело в том, что биллинговый сервис должен обработать буферизованные события до того, как он начнет обрабатывать входящий трафик, но производитель сервиса оформления заказов все еще выдает события. Через некоторое время все буферизованные события будут потреблены и весь бизнес-процесс двух приложений продолжит обработку с типичной задержкой и трафиком.

Аналогичное решение можно применить при неожиданном пике трафика. Представьте ситуацию, в которой сервис оформления заказов начинает производить свои 200 событий в секунду. Сервис биллинга не справится с трафиком, потому что он предлагает условия SLA в 100 запросов в секунду. Но если эта ситуация временная, дополнительные события будут буферизоваться в топике Kafka. Когда трафик возвратится к нормальному уровню, биллинговый сервис сможет обработать дополнительный трафик и через какое-то время вернется к обычному объему.

Вы можете встраивать эту форму отказоустойчивости в системы при условии, что вы используете архитектуру pub-sub и компонент, способный буферизовать трафик. Также необходимо иметь возможность анализировать гарантии доставки этих компонентов. Далее остается выбрать семантику доставки, соответствующую потребностям.

Второй аспект, критичный для этого решения, — возможность обработки дополнительного трафика потребителем. Если условия SLA потребителя не будут

значительно выше, чем у производителя, время возобновления обработки будет большим. Потребитель должен иметь возможность обрабатывать буферизованный трафик наряду с нормальным входящим трафиком. В противном случае он не справится с трафиком при частичном сбое. А теперь подведем итоги главы.

ИТОГИ

- Архитектура «публикация-подписка» (pub-sub) позволяет создавать асинхронные системы со слабой связанностью, улучшающие отказоустойчивость с соответствующими гарантиями доставки.
- Очередь событий предоставляет средства для создания событийно-управляемых архитектур. Чем больше сервисов взаимодействует друг с другом, тем больше преимуществ от использования этой архитектуры.
- Можно анализировать семантику доставки и управлять ею как на стороне производителя, так и на стороне потребителя.
- В распределенных системах можно выдерживать баланс между целостностью данных и доступностью.
- Код потребителя может быть реализован с гарантиями доставки Kafka «не менее одного» и «не более одного».
- Детализированные фиксации позволяют сократить количество дубликатов с гарантией «не менее одного» на стороне потребителя.
- Гарантия «не более одного» создает риск того, что в случае сбоя будут обработаны не все события.
- При разбиении функциональности очереди на N -независимые системы наши системы могут одновременно обладать слабой связанностью и асинхронной отказоустойчивостью без единой точки сбоя.
- Гарантия «фактически ровно один» возможна при использовании транзакций. Однако в более сложных архитектурах с множеством сервисов в конвейере она становится слишком сложной или невыполнимой.

12

Управление версиями и совместимостью

В этой главе

- ✓ Версионирование на абстрактном уровне.
- ✓ Планирование стратегий версионирования для библиотек.
- ✓ Проектирование API для дальнейшей эволюции.
- ✓ Эффективная работа со схемами хранения.

Есть темы, которые у опытного инженера наверняка вызовут приступ зубной боли. Если в беседе вы заговорите о локализации, конфликтах слияния или часовых поясах, будьте готовы к холодному приему. То же касается и версионирования. Это неотъемлемая часть процесса разработки, но многие специалисты не торопятся разбираться с ним и постоянно откладывают эту задачу — отчасти потому, что она считается напрасной тратой времени. Когда продукт, библиотека или API *правильно* подходят к версионированию, на этот аспект редко обращают внимание, но при *некачественной* реализации он может вызвать массу претензий.

В этой главе мы представим некоторые точки зрения на версионирование, которые помогут вам разработать стратегию, наиболее подходящую для своего продукта. Мы дадим конкретные рекомендации и предложения, но в конечном счете ответственность за соблюдение баланса между противоречащими друг другу факторами и техническими проблемами ляжет на вас.

Впрочем, самый главный совет легко дать уже сейчас: не отворачивайтесь в надежде, что без версионирования удастся обойтись.

Если только вы не собираетесь выбросить свой код в мусорное ведро почти сразу после написания, вам придется планировать его развитие и оценивать, к каким последствиям это может привести. Прежде чем погружаться в специфику библиотек, сетевых API (таких, как веб-сервисы) и хранения данных, давайте разберемся, что мы называем версионированием (versioning) и почему оно вообще существует.

12.1. ВЕРСИОНИРОВАНИЕ НА АБСТРАКТНОМ УРОВНЕ

Все меняется. Если вы ищете область, в которой вам никогда не придется иметь дела с новыми задачами или требованиями, скорее всего, программирование вам не подойдет. Изменения многогранны; вероятно, самое очевидное — смена набора требований к конкретному блоку кода, но на практике практически каждый аспект того, что вы используете (оборудование, операционная система, платформа разработки, язык программирования, модель развертывания — да все что угодно), меняется со временем. Все эти изменения могут создать непредвиденные сложности.

Версионирование пытается взять под контроль эту сложность, выступая средством коммуникации между людьми и системами по части ожиданий. Разные схемы версионирования адаптируются к разным требованиям и ожиданиям; в этом разделе мы узнаем, что у них общего и чем они различаются.

12.1.1. Свойства версий

Версии имеются у самых разных сущностей: приложений, библиотек, протоколов, книг, языков программирования и т. д. Впрочем, у разных схем версионирования разные свойства, и мы выясним, по каким параметрам они различаются.

Запоминаемость

Многие схемы версионирования разрабатывались так, чтобы быть хорошо читаемыми и запоминаемыми; трудно забыть, что вы используете, например, Ubuntu 20.04 или что вы читаете первое издание этой книги. Некоторые схемы версионирования практически *не запоминаются* — самым ярким тому примером являются хеши git. Не уверен, что я когда-нибудь запомню af257385d785f597fc8be67c84f2cf714fbe4203, и даже af25738 (первые семь символов, выводимые в GitHub, Bitbucket и т. д.) запомнить непросто, если приходится работать с несколькими коммитами.

Неизменяемость

Часто (но не всегда) версии, от которых мы программно зависим, довольно точны и неизменяемы. К примеру, версия 3.0.4 NuGet-пакета NodaTime из поставки nuget.org всегда состоит из тех же байтов. Канал пакетов предотвращает перезапись существующей версии.

Хеши git скорее неизменяемы по своей природе: хеш вычисляется по содержимому и при изменении содержимого *обязан* поменяться (не считая астрономически маловероятного события хеш-коллизии).

Это весьма полезное свойство, которое делает программные системы более предсказуемыми. Впрочем, для человека оно менее полезно. Если спросить кого-нибудь, какую операционную систему он использует, ответом скорее будет *Windows 10*, чем *Windows 10, сборка 19042.867*.

Неизменяемость и запоминаемость не полные противоположности, но между ними существуют определенные трения, так как неизменяемые номера версий обычно более информативны.

Неявные отношения между версиями

Многие системы версионирования пытаются передать важную информацию в минимальном объеме данных (текст номера версии). Самый очевидный пример рассматривается в разделе 12.1.3 при описании семантического версионирования, но есть и множество других примеров. Visual Studio использует маркетинговые версии с упоминанием года: Visual Studio 2019 очевидно более поздняя версия, чем, скажем, Visual Studio 2017.

Некоторые версии не содержат никакой информации о порядке или отношениях между ними. И снова очевидный пример — это хеши git. Если вы получите два хеша git без доступа к самому репозиторию, вы не сможете сказать, является ли один из них частью истории другого или это две независимые ветви, и даже находятся ли они в одном репозитории.

Наконец, некоторые схемы версионирования на первый взгляд содержат эту информацию, но с ними следует быть осторожными. Как вы думаете, что было раньше: Xbox или Xbox One? И когда ожидать Xbox 360? История версий .NET тоже *интересна*, но вы вряд ли захотите использовать подобную модель.

При создании собственных схем версионирования или выборе из существующих вариантов подумайте, что вы хотите донести и какую информацию потребитель схемы сможет вывести самостоятельно, даже если это не входило в ваши намерения.

Важная часть информации, которую многие схемы версионирования пытаются предоставить в тексте версии, — гарантия *совместимости* (или предположение о несовместимости). Разберемся, что же понимается под совместимостью.

12.1.2. Обратная и прямая совместимость

В широком смысле совместимость — это про то, что произойдет, если одну версию продукта использовать с другой версией того же продукта. Формулировка намеренно расплывчатая, потому что эта концепция может применяться по-разному.

Обратная совместимость — свойство, означающее, что новая версия может работать с информацией из старой. *Прямая совместимость* — свойство, означающее, что старая версия может работать с информацией из новой.

Конкретные примеры этих концепций:

- Java как язык поддерживает *обратную совместимость*. Код, написанный на Java 7, может быть откомпилирован компилятором Java 17. При этом *прямая совместимость* не сохраняется: в коде, написанном на Java 17, могут быть возможности, которые приведут к ошибке компиляции при использовании компилятора Java 7.
- Библиотеки обычно пишутся с учетом обратной совместимости; код, написанный для NodaTime версии 2.3, будет работать с NodaTime версии 2.4. Версионирование библиотек намного более подробно рассматривается в разделе 12.2. Версии исправлений в рамках семантического версионирования обеспечивают прямую совместимость, как будет показано в следующем разделе.
- Веб-сервисы обычно поддерживают обратную совместимость; запрос к веб-сервису, закодированный в формате JSON и написанный для определения сервиса от 10 января 2021 года, должен работать 1 апреля 2021 года, даже если определение сервиса изменилось. Впрочем, вызывающему коду, возможно, придется обрабатывать ожидаемые данные в ответе (что обусловлено совместимостью данных). Версионирование сетевых API более подробно рассматривается в разделе 12.3.
- Некоторые форматы данных — например, Google Protocol Buffers и Apache Avro — спроектированы для обеспечения как обратной, так и прямой совместимости, чтобы старый код мог работать с новыми хранилищами данных без потери данных, записанных новым кодом. Версионирование данных рассматривается в разделе 12.4.

Иногда термины «прямая» и «обратная» создают больше путаницы, чем пользы и приводят к недоразумениям на совещаниях. На мой взгляд, лучше вести обсуждение с конкретными примерами; вместо того чтобы говорить о *новой версии клиента*, укажите конкретные номера версий. Это не обязательно должны быть реальные номера версий продукта, который вы выпустили или собираетесь выпустить, это могут быть гипотетические числа. Однако конкретика помогает избежать путаницы. По аналогии с паттерном «дано — если — то» в примечном тестировании эти сценарии начинаются с определения характеристик

конкретных версий, представления конкретного набора взаимодействий и попытки воспроизвести результаты. Несколько примеров такого рода встречаются далее в главе.

12.1.3. Семантическое версионирование

Семантическое версионирование (часто обозначаемое сокращением SemVer) стало самым популярным подходом в библиотечных экосистемах большинства платформ (хотя бы в теории). Полное соблюдение правил семантического версионирования встречается, мягко говоря, нечасто; почему — читайте в разделе 12.2.4.

Правила для стабильных версий

Номер версии по правилам семантического версионирования всегда состоит из трех компонентов: номера основной версии, номера дополнительной версии и номера исправления. На рис. 12.1 номер основной версии равен 2, дополнительной — 13, а номер исправления — 4.



Рис. 12.1. Пример семантической версии

Основные правила семантического версионирования (при применении к одной сущности — например, библиотеке):

- если две версии различаются по основному компоненту, то гарантии их совместимости нет. Например, версии 2.13.4 и 3.0.2 могут быть полностью несовместимы;
- если основной компонент двух версий одинаков, но дополнительные версии разные, то версия с большим дополнительным номером должна обладать обратной совместимостью с версией с меньшим дополнительным номером. Например, версия 2.13.4 должна обладать обратной совместимостью с 2.5.3;
- если у двух версий одинаковые основные и дополнительные компоненты, они должны обладать обратной и прямой совместимостью друг с другом. Например, версии 2.13.4 и 2.13.1 должны быть совместимы в обоих направлениях.

Эти правила разрабатывались для понимания того, когда потребители могут изменять используемую версию:

- при изменении основной версии все гарантии пропадают. Действуйте осторожно, проводите тщательное тестирование и выделите больше времени на обновление;

- если вы переходите на более позднюю дополнительную версию, все *должно быть* нормально (хотя иногда это может быть не так, несмотря на все усилия, как будет показано дальше);
- при переходе или *возврате* к исправлениям с одинаковыми номерами дополнительных версий все должно быть в порядке. Однако следует учитывать, что версии исправлений обычно существуют для исправления багов, и может оказаться, что вы зависите от этого багазованного поведения. (Например, в приложении могло использоваться обходное решение, работоспособность которого нарушается из-за отладки бага.) Возврат к предыдущей версии должен пройти нормально.

При обсуждении конкретных версий в качестве заменителей любого числа часто используются x и y . Например, можно сказать, что версия 1.3. x должна обладать обратной совместимостью с 1.2. y .

Нестабильные версии

Семантическое версионирование также предоставляет два механизма для обозначения нестабильных версий. В ходе начальной разработки используется основная версия 0. Обычные предположения семантического версионирования неприменимы к основной версии 0; так, версия 0.2.0 может быть полностью несовместима с версией 0.1.0. Даже смена версии исправления (скажем, переход с 0.1.0 на 0.1.1) не обязательно сохранит совместимость.

Второй способ обозначения нестабильной версии основан на использовании предвыпускной метки. В этом случае в обычной семантической версии добавляется суффикс из дефиса, за которым следует последовательность идентификаторов, разделенных точками.



Рис. 12.2. Пример семантической версии с предвыпускной меткой

На рис. 12.2 часть «Основная.Дополнительная.Исправление» равна 1.4.6 и есть предвыпускная метка beta.1. Предвыпускная метка может содержать сколько угодно идентификаторов, разделенных запятыми, но обычно включает от 1 до 3. Каждый из них должен состоять из алфавитно-цифровых ASCII-символов или дефисов.

И хотя гарантий совместимости с предвыпускными версиями нет, обычно их номера указывают на движение к версии, обозначаемой частью «Основная.Дополнительная.Исправление», которая предполагает совместимость с другими версиями. Например, версия 1.5.0-alpha.1 по этой логике должна обладать обратной совместимостью с 1.4.x.

ВЫБОР МЕЖДУ 0.x.y И ПРЕДВЫПУСКНЫМИ МЕТКАМИ

Хотя исторически номера версии 0.x.y были более распространены, у них есть значительный недостаток: они работают только для первой стабильной версии (1.0.0). Если исходная последовательность версий имела вид 0.8.0, 0.9.0, 1.0, появляется соблазн использовать версии 1.8, 1.9, 2.0 для второй основной версии. В большинстве случаев это приведет к нарушению правил семантического версионирования, потому что переход на версию 2.0 обычно вводит несовместимые изменения.

Я рекомендую использовать предвыпускные метки с самого первого открытого релиза. После этого можно повторить последовательность выпусков первой основной версии (например, 1.0.0-alpha.1, 1.0.0-beta.1, 1.0.0-beta.2, 1.0.0) и второй основной версии (2.0.0-alpha.1, 2.0.0-alpha.2, 2.0.0-beta.1, 2.0.0). Обозначения 0.x.y лучше зарезервировать для очень ранних прототипов или полностью избегать таких номеров версий.

Метаданные сборки

Семантическое версионирование также позволяет указать *метаданные сборки* после стабильной или предвыпускной версии с суффиксом «+». Метаданные представляют собой последовательность идентификаторов, сходную с предвыпускными метками. Предполагается, что они используются только как дополнительная информация. Например, в них можно включить метку времени или хеш сохранения в репозитории.

На рис. 12.3 часть «Основная.Дополнительная.Исправление» содержит 1.2.3, предвыпускная метка — beta.1, а метаданные сборки — 20210321.af25738. В данном случае метаданные сборки включают дату и хеш сохранения, но для семантического версионирования это не имеет значения.

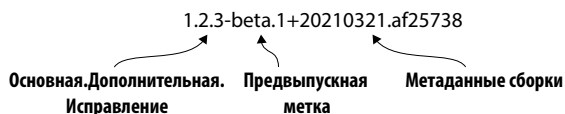


Рис. 12.3. Пример семантической версии с предвыпускной меткой и метадаанными сборки

Отношения предшествования между версиями

Семантическое версионирование определяет отношения *предшествования* между версиями. Эти отношения используются в инструментах для определения совместимости там, где это возможно, и предполагаемых обновлений. В общем случае они устанавливаются предсказуемо:

- 1.2.3 предшествует 2.0.0;

- 1.2.3 предшествует 1.3.0;
- 1.2.3 предшествует 1.2.4;
- 1.3.0-alpha.5 предшествует 1.3.0-beta.1;
- 1.3.0-beta.8 предшествует 1.3.0-beta.10 (обратите внимание на сравнение чисел);
- 1.3.0-beta.2 предшествует 1.3.0.

Полные и точные правила предшествования выходят за рамки книги, но их можно найти на странице <https://semver.org>. Напоследок перейдем от семантического версионирования с его точными правилами к другой крайности: маркетингу.

12.1.4. Маркетинговые версии

Семантическое версионирование разрабатывалось для компактной передачи технических данных. У него нет задачи мотивировать к покупке нового продукта. Для этого предназначены маркетинговые версии. Я упоминаю их в этой главе только по одной причине: чтобы подчеркнуть различия между маркетинговым и семантическим версионированием.

Чаще всего маркетинговые версии не нужны. Обычно они создаются для продуктов, а не для библиотек, протоколов, форматов файлов или схем. Там, где возникает потребность в маркетинговых версиях, часто также существует более техническая версия, используемая прежде всего для целей поддержки. Как правило, она более длинная и связана (или нет) с семантическим версионированием. Она может даже противоречить маркетинговой версии. Можно, к примеру, выпустить потрясающую игру *Awesome Game* с финальной технической версией 2.3.1, а потом выпустить *Awesome Game 2* с исходной технической версией 1.0.0.

Необходимо понять главное: это разные схемы версионирования, которые служат совершенно разным целям. Постарайтесь их не путать — как при просмотре имеющихся версий, так и при проектировании собственных схем.

До сих пор мы рассматривали версионирование на абстрактном уровне, хотя семантическое версионирование применяется *в первую очередь* для библиотек. Тема версий библиотек более подробно разбирается в следующем разделе.

12.2. ВЕРСИОНИРОВАНИЕ ДЛЯ БИБЛИОТЕК

Для многих разработчиков библиотеки составляют самую важную область версионирования. Впрочем, если вы только *пользуетесь* библиотеками, все должно быть довольно просто — или казаться простым. Работать с разными версиями

набора библиотек всегда сложна и может создать массу проблем. Даже если все разработчики всех библиотек, которыми вы пользуетесь, соблюдают соглашения для вашей платформы (как правило, семантическое версионирование), нет гарантий, что вам удастся избежать проблем несовместимости. Тем не менее выбор возможных решений относительно невелик.

Большая часть этого раздела предназначена для разработчиков, *публикующих* библиотеки. Им приходится принимать гораздо больше решений, многие из которых направлены на соблюдение баланса на основании разумных предположений. Даже в самой концепции публикации библиотеки есть нюансы. Приходится учитывать разные обстоятельства при публикации в менеджере пакетов (таком, как Maven Central или NuGet) или во внутреннем репозитории компании, а также просто обновляя исходный код библиотеки для потребления другим исходным кодом.

Простых решений не бывает, но эти рекомендации помогут задать вопросы в нужном контексте и найти *наименее худшие* возможные ответы. А если учесть, что версионирование библиотек во многом ориентировано на совместимость, для начала стоит понять, что это значит.

12.2.1. Совместимость уровня исходного кода, двоичная и семантическая

Рассмотрим одну частую ситуацию: требуется опубликовать новую версию библиотеки, и вы хотите знать, обладает ли она обратной совместимостью со старой версией этой же библиотеки. Будем считать, что мы ничего не знаем о коде, который потребляет нашу библиотеку, и каждый пример станет своего рода испытанием: удастся ли нам придумать *гипотетический* код потребления, работоспособность которого нарушится нашими изменениями? После этого можно проанализировать природу этих нарушений.

ПРИМЕЧАНИЕ

Под термином «код потребления» в этом разделе понимается код приложения (или другой библиотеки), который зависит от рассматриваемой библиотеки. Часто это означает, что он вызывает функции, предоставляемые библиотекой, но не всегда. Например, код потребления может просто реализовывать интерфейс, предоставляемый библиотекой.

Внимание, спойлер: почти для любого теоретически возможного изменения можно придумать код потребления, работоспособность которого будет нарушена этим изменением. Яркий пример: код потребления может получать хеш-код библиотеки и выдавать исключение, если он отличается от ожидаемого. В таком случае изменить библиотеку *никак* нельзя без нарушения работоспособности этого потребителя.

К счастью, код потребления обычно ведет себя более разумно. И все равно могут возникнуть неочевидные ситуации, когда изменение нарушит его работоспособность. Что делать: признать такое изменение критичным, соглашаясь на сопутствующие затраты для других потребителей, или посчитать его слишком маловероятным и игнорировать? Такие решения придется принимать, когда вы выясните, какой код может быть нарушен тем или иным изменением.

Для компилируемых языков следует учитывать три вида совместимости: совместимость исходного кода, двоичную и семантическую совместимость. Концепция двоичной совместимости обычно неприменима к языкам с предварительной компиляцией, где библиотека фактически публикуется в виде исходного кода. Например, это относится к библиотекам JavaScript, таким как React и jQuery. Все приведенные ниже примеры написаны на Java, но правила совместимости сильно зависят от языка; наша цель — показать ход мыслей, не концентрируясь на конкретных изменениях.

В большинстве примеров приводится код библиотеки до и после изменений, а также код потребления, использующий библиотеку. Начнем с совместимости исходного кода.

Совместимость исходного кода

Новая версия библиотеки обладает *совместимостью исходного кода* с более ранней версией, если код потребления, который работал со старой версией, также нормально работает с более поздней версией. В Java код потребления необходимо перекомпилировать для новой библиотеки. Начнем с очевидного примера несовместимого изменения: переименования метода, пусть даже лишь с изменением регистра одной буквы, как показано в следующем листинге.

Листинг 12.1. Переименование метода

Библиотека до изменения

```
public static User getByID(int userId) {
    ...
}
```

Библиотека после изменения

```
public static User getById(int userId) {
    ...
}
```

Код потребления

```
int userId = request.getUserId();
User user = User.getByID(userId);
```

После изменения библиотеки возникнет ошибка компиляции кода потребления «не удастся найти символическое имя для `User.getByID`». Переименование *чего угодно* общедоступного (пакета, поля, интерфейса, класса или метода) является критическим изменением. Однако не каждое критическое изменение настолько очевидно. Возьмем преобразование параметра одного типа к супертипу — например, `String` к `Object`, как показано в следующем листинге.

Листинг 12.2. Изменение типа параметра (совместимость исходного кода)

Библиотека до изменения

```
public void displayData(String data) {
    ...
}
```

Библиотека после изменения

```
public void displayData(Object data) {
    ...
}
```

Любой код, *вызывающий* метод, будет работать нормально. Даже преобразование ссылки на метод в коде потребления должно выполняться. Однако код потребления может не ограничиться простым вызовом метода.

Код потребления

```
public class ConsumerClass extends LibraryClass {
    @Override
    public void displayData(String data) {
        ...
    }
}
```

С исходной версией все было в порядке, но измененный код библиотеки приводит к ошибке компиляции. Если класс, содержащий метод, был помечен как финальный, это запрещает создание подклассов; *возможно*, в этом случае изменение обладает совместимостью исходного кода. В целом следует крайне внимательно относиться к любому изменению открытого API, когда нет полной уверенности в сохранении совместимости.

ПРИМЕЧАНИЕ

Хотя приведенные примеры написаны на Java, общая концепция совместимости применима к другим языкам. Следует помнить, что изменение, сохраняющее совместимость исходного кода или двоичную совместимость на одном языке, *не всегда* будет совместимым в другом. Например, переименование параметра метода сохраняет обратную совместимость в Java, но не в C# из-за поддержки *именованных аргументов* в C#. Дополнительная сложность в том, что правила относительно того, какие изменения сохраняют обратную совместимость, могут со временем меняться. (Например, в C# не всегда существовали именованные аргументы, а в Java они могут появиться в будущем.)

Как правило, добавление чего-то нового считается совместимым изменением, хотя *теоретически* оно может нарушить работоспособность исходного кода потребления за счет создания конфликтов имен. Один важный контрпример — добавление методов в интерфейсы; если только вы не предоставите реализацию метода по умолчанию, такое изменение будет критическим, так как любой потребитель, объявляющий, что он реализует интерфейс, теперь будет реализовывать его *не полностью*. То же относится к добавлению абстрактного метода в существующий абстрактный класс.

До сих пор мы рассматривали ситуацию, в которой библиотека изменяется, а затем код потребления перекомпилируется. Что произойдет, если перекомпиляция кода потребления невозможна? В такой ситуации совместимость называется *двоичной*.

Двоичная совместимость

Прежде чем углубляться в то, какие изменения обладают двоичной совместимостью, а какие — нет, стоит отметить, почему это важно. В конце концов, мы ведь обычно перекомпилируем свои приложения при необходимости? И хотя на уровне приложения такой подход работает, как правило, он менее приемлем для других зависимостей. Эти сложности будут рассматриваться при описании графов зависимостей в разделе 12.2.2, но представьте следующую ситуацию:

- приложение зависит от библиотек LibraryA и LibraryB;
- LibraryA *также* зависит от LibraryB.

Схема ситуации приведена на рис. 12.4, где каждая стрелка представляет зависимость. Если LibraryB вносит изменение, имеющее после перекомпиляции лишь обратную совместимость, также необходимо использовать перекомпилированную версию LibraryA, а с этим могут возникнуть проблемы.

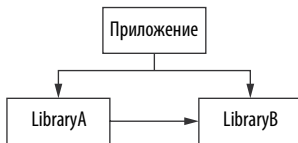


Рис. 12.4. Простое приложение с двумя зависимостями от LibraryB

Двоичную совместимость анализировать труднее, чем совместимость исходного кода, потому что она подразумевает уровень абстракции, который разработчик обычно игнорирует. Так, многим разработчикам Java не нужно знать, какой байткод будет сгенерирован для конкретного блока кода. К счастью, в спецификации языка Java есть целый раздел о том, какие изменения обладают двоичной совместимостью, а какие нет. Однако не стоит ожидать, что подобная документация будет доступна для каждого языка!

Некоторые изменения, несовместимые на двоичном уровне, очевидны: удаление или переименование методов или типов очевидно создаст проблемы. В других случаях все не так однозначно. Вернемся к примеру с изменением типа параметра метода со `String` на `Object`. Если не обращать внимания на проблему с переопределением, рассмотренную ранее, совместимости исходного кода достичь можно, поскольку существует неявное преобразование `String` в `Object`. Впрочем, компилятору известно об этом преобразовании. Во время выполнения JVM ожидает, что сигнатура метода останется неизменной. Рассмотрим конкретный пример.

Листинг 12.3. Изменение типа параметра (двоичная совместимость)

Библиотека до изменения

```
public void displayData(String data) {
    ...
}
```

Библиотека после изменения

```
public void displayData(Object data) {
    ...
}
```

Код потребления

```
public class Program {
    public static void main(String[] args) {
        new LibraryClass().displayData("Hello");
    }
}
```

Если попытаться выполнить код потребления, скомпилированный для исходного кода библиотеки, перекомпилировав только ее измененный код, во время выполнения будет выдана ошибка:

```
java.lang.NoSuchMethodError: 'void LibraryClass.displayData(java.lang.String)'
```

Реализацию двоичной совместимости усложняют два обстоятельства:

- *проблема проявляется лишь во время выполнения.* Разработчики, использующие компилируемые языки, привыкли, что компилятор выявляет подобные проблемы (отсутствие необходимого метода);
- *проблема проявляется только в том случае, если выполнение идет по ветви, которая ищет отсутствующий метод.* И это особенно неприятно, так как ветви кода с обработкой ошибок обычно тестируются менее тщательно, что только повышает вероятность ошибок.

Как вы уже видели, некоторые изменения поверхности API могут быть совместимыми по исходному коду, но несовместимыми на двоичном уровне. Такие изменения, как добавление нового класса, могут обладать двоичной совместимостью,

но формально быть несовместимыми по исходному коду (из-за конфликтов имен). Другие изменения либо обладают и совместимостью исходного кода, и двоичной совместимостью, либо несовместимы на обоих уровнях.

Впрочем, все сказанное до сих пор в основном относилось к открытым API; мы не углублялись в подробности реализации. Однако «код компилируется и все методы найдены» и «все работает как прежде» — не одно и то же. Последняя совместимость, которую мы рассмотрим, — семантическая совместимость.

Семантическая совместимость

Двоичная совместимость довольно проста. У исходной совместимости больше нюансов; нужно решить, считать ли конфликты имен критическим изменением. Впрочем, обычно все достаточно ясно. Семантическая совместимость, как правило, тесно связана с поведением кода, и вы не знаете, от чего люди будут зависеть. Этот принцип выражен в законе Хайрама (по имени Хайрама Райта, инженера-разработчика в Google; см. <https://www.hyrumslaw.com/>):

При достаточном количестве пользователей API становится не так важно, что вы обещаете в контракте: от каждого наблюдаемого поведения системы что-то да будет зависеть.

Если свести этот принцип к одной крайности, каждый выпуск библиотеки будет иметь новую основную версию в семантическом версионировании. В другой крайности можно выбрать позицию «Если вам не нравится новое поведение, не пользуйтесь библиотекой» и рассматривать любое изменение, не влияющее на поверхность открытого API, как некритическое. Конечно, ни одна из крайностей не оптимальна.

Очевидно, многие изменения реализации не приводят к значительным изменениям поведения. Впрочем, существуют три класса изменений, которые заслуживают особого внимания:

- проверка параметров;
- наследование;
- изменения производительности.

Изменения в проверке параметров обычно делятся на две подкатегории. Ввод, который должен быть признан недействительным, был случайно принят, и вы хотите либо усилить проверку для его отклонения (как исправление ошибки), либо разрешить вывод, который отклонялся ранее (как расширение функциональности). В качестве примера второго случая рассмотрим простой класс `Person` для хранения полного имени и *неформального* имени (прозвища, уменьшительной формы и т. д.). Изначально допустимы обе формы, как показано в следующем листинге.

Листинг 12.4. Исходный класс `Person` с двумя параметрами, не допускающими `null`

```
public class Person {
    private final String legalName;
    private final String casualName;
    public Person(String legalName, String casualName) {
        this.legalName = Objects.requireNonNull(legalName);
        this.casualName = Objects.requireNonNull(casualName);
    }
}
```

Если `casualName` содержит `null`,
выдается исключение

Предположим, мы обнаружили, что на самом деле многим пользователям не нужно отдельное неформальное имя. Они постоянно передают одинаковые значения для официального и неформального имени, и иногда это заметно усложняет код. Можно изменить библиотеку, чтобы она допускала передачу `null`, а по умолчанию одно значение использовалось для обоих полей.

Листинг 12.5. Изменение конструктора класса `Person` для передачи `null` вместо неформального имени

```
public class Person {
    private final String legalName;
    private final String casualName;
    public Person(String legalName, String casualName) {
        this.legalName = Objects.requireNonNull(legalName);
        this.casualName = casualName == null ? casualName : legalName;
    }
}
```

`casualName` может
содержать `null`,
а `legalName` используется
в исходном виде

В каком-то смысле это изменение является совместимым. Но если потребитель полагался на проверку параметра `casualName`, его ждет неприятный сюрприз. И что еще хуже, ошибка может быть *незаметной*. Возьмем следующий метод потребителя, который в данном случае выводит информацию на экран, но с таким же успехом мог бы создавать HTML-разметку для веб-страницы или делать что-то подобное:

```
public static void createUser(String legalName, String casualName) {
    Person person = new Person(legalName, casualName);
    System.out.println("Welcome, " + casualName);
    ...
}
```

Использовать `Person`
для дальнейших операций

Предполагается,
что `casualName`
проверяется на `null`

Если параметр `casualName` метода `createUser` содержит `null`, этот код выведет сообщение «*Welcome, null*» вместо исключения, как было до этого. Но параметр может использоваться в других местах метода, распространяя значение `null` там, где раньше оно не предполагалось. Этот код может быть включен в библиотеку с документированием проверки `casualName`, которая должна выполняться в конструкторе `Person`.

Для этого случая существует альтернативное решение с обратной совместимостью — добавление конструктора, получающего только одно имя (которое затем используется и как официальное, и как неформальное). Если выясняется, что

необходимо ослабить (или усилить) проверку, добавление альтернативного пути (через перегрузку или добавление нового метода) поможет избежать незаметных критических изменений.

Наследование может вызвать семантические изменения, когда то, что кажется подробностью реализации, фактически раскрывается из-за возможности переопределения методов. В листинге 12.6 приведены классы `Player` и `Position`, которые могут присутствовать в исходном коде игры.

Листинг 12.6. Исходные версии классов `Player` и `Position`

```
public final class Position {
    private final int x;
    private final int y;
    public Position(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
public class Player {
    private Position position;
    public void moveTo(int x, int y) { ← Делегирование методу,
        moveTo(new Position(x, y));      получающему Position
    }
    public void moveTo(Position position) {
        this.position = position; ← Настоящие изменения в методе,
    }                                     получающему Position
    ...
}
```

Допустим, подкласс `Player` хочет ограничить перемещение игрока конкретной областью. Для этого он может переопределить метод `moveTo(Position)`, чтобы тот задавал ограниченную позицию, а затем вызывал `super.moveTo(actualPosition)` для завершения операции. Однако автор класса `Player` может решить, что постоянно создавать объекты `Position` нежелательно и лучше передавать значения `x` и `y` напрямую. Он полагает, что можно внести изменение с обратной совместимостью простой заменой делегирования в методах `moveTo`, как показано в листинге 12.7.

Листинг 12.7. Измененный класс `Player` с заменой делегирования

```
public class Player {
    private int x;
    private int y;
    public void moveTo(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void moveTo(Position position) {
        moveTo(position.getX(), position.getY()); ← Делегирование методу
    }                                             с координатами
}
```

Координаты хранятся в виде целых чисел

Метод с координатами становится основной реализацией

В этой точке подкласс ведет себя правильно только при вызове перегруженной версии `moveTo(Position)`. Она будет ограничивать ввод и вызывать реализацию из `Player`, которая затем делегирует перегруженной версии `moveTo(int, int)`. Но если пользователь вызывает `moveTo(int, int)` напрямую, выполнение обходит ограничивающий код.

Это было бы приемлемо, если бы класс `Player` или методы `moveTo` были финальными. Но наследование и возможность наследования некоторых методов фактически раскрывает подробность реализации (то, какая перегруженная версия вызывает другую перегруженную версию).

Данный пример приводит нас к последней категории семантических изменений, также заслуживающих внимательного рассмотрения: изменений производительности. (Хотя вопрос, можно ли считать их семантическими, остается открытым. Это наблюдаемое изменение поведения, даже если поведение не сводится к *вводу и выводу*.) В продолжение примера с классом `Player` мы отмечали, что исходное изменение должно было предотвратить создание многочисленных объектов `Position`. Однако в зависимости от того, как используется класс, эффект может оказаться обратным. Допустим, у класса `Player` имеются методы доступа как для позиции, так и для ее составляющих. Они могут быть реализованы двумя способами, как показано в следующем листинге.

Листинг 12.8. Методы доступа в классе `Player`

Методы доступа до изменения

```
public int getPositionX() {
    return position.getX();
}
public int getPositionY() {
    return position.getY();
}
public Position getPosition() {
    return position;
}
```

Методы доступа после изменения

```
public int getPositionX() {
    return x;
}
public int getPositionY() {
    return y;
}
public Position getPosition() {
    return new Position(x, y);
}
```

Повысится ли производительность класса `Player` после такого изменения? Ответ полностью зависит от способа использования класса. Код с вызовами `moveTo(int, int)`, `getPositionX()` и `getPositionY()` определенно будет выполнять меньше операций выделения памяти, но в коде с вызовами `getPosition()` их станет больше. Фактически изменяется лучший способ использования библиотеки: прежняя схема, эффективная по выделению памяти, стала неэффективной, и наоборот. Если для библиотеки производительность очень важна, это изменение можно считать критическим из-за необходимости менять вызывающий код.

Итак, теперь вы можете оценить степень, в которой любое заданное изменение сохраняет или не сохраняет обратную совместимость. Значит ли это, что тема исчерпана? *Можно* прийти к выводу, что библиотека требует лишь следования правилам семантического версионирования на основании результатов проверки каждого изменения. Можно сказать себе: «Ок, если понадобится внести критическое изменение, мы просто расширим основную версию, и все потребители поймут, что это значит». Если бы все приложения только пользовались библиотеками без дальнейших зависимостей, все было бы действительно просто. Добавление критического изменения создало бы *некоторые* неудобства для потребителей (им пришлось бы проверить, что оно не влияет на их код, или внести необходимые изменения), но это не так страшно.

К сожалению, в жизни все не так просто. Давайте выясним, что происходит при использовании множества библиотек, когда графы зависимостей начинают разрастаться.

12.2.2. Графы зависимостей и ромбовидные зависимости

Должен предупредить, что это весьма скользкая тема. Иногда, видя очень крупный граф зависимостей, я поражаюсь, как сильно мы полагаемся на наше ПО и на то, что оно продолжит работать в процессе своего развития. Проблемы, описанные в этом разделе, абсолютно реальны; это подтвердит каждый, у кого есть печальный опыт борьбы с конфликтами зависимостей. Но, как правило, мы с ними справляемся. На помощь приходит синяя изолента.

ПРИМЕЧАНИЕ

Номера версий библиотек приводятся в этом разделе для примера. Будем считать, что во всех библиотеках, о которых идет речь, используется семантическое версионирование. Если в графе зависимостей встречаются библиотеки, которые его не используют, это ничего принципиально не меняет, а только намного усложняет рассуждения.

До сих пор мы рассматривали примеры, в которых приложение зависит от одной библиотеки, и не учитывали зависимости, которые может иметь эта библиотека. Теперь рассмотрим ситуации, в которых приложение зависит от ряда библиотек.

Каждая из них может иметь свои зависимости, а те в свою очередь — несколько своих. Эти зависимости представляются в виде направленного графа: каждый узел представляет одну библиотеку (со всеми версиями), а каждая стрелка — отношение зависимости, которое помечается версией этой зависимости.

ПРИМЕЧАНИЕ

Некоторые программные инструменты (в том числе Maven) для представления зависимостей используют деревья вместо графов. Каждый узел дерева состоит из артефакта и номера его версии (вместо ребра графа, показывающего версию зависимости). Обе формы представляют одинаковую информацию, но мне проще выявлять ромбовидные зависимости на графах.

Если все это звучит слишком запутанно, не беспокойтесь: на схеме все намного понятнее. (Проблема больших графов зависимостей не из простых, но разобраться в самом графе не так сложно.)

Рассмотрим гипотетический пример. Есть приложение, которое должно читать файлы JSON; в нем используется очередь сообщений и база данных. При этом библиотека очереди сообщений также использует функциональность JSON. Зависимости могут выглядеть примерно так:

- приложение зависит от JsonLib версии 1.2.0;
- приложение зависит от MQLib версии 2.1.2;
- приложение зависит от DbLib версии 3.5.0;
- MQLib зависит от JsonLib версии 1.1.5.

С точки зрения приложения последний пункт представляет *транзитивную зависимость*. Это косвенная зависимость, которая присутствует здесь только из-за того, что приложение зависит от MQLib. Транзитивные зависимости могут включать библиотеки, от которых основное приложение не зависит напрямую.

Граф для этого набора зависимостей изображен на рис. 12.5.

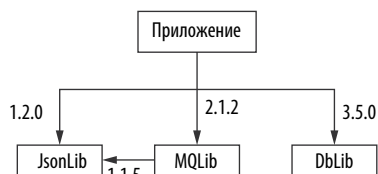


Рис. 12.5. Граф зависимостей простого приложения

Хотя этот гипотетический пример намного проще многих реальных приложений, в которых общий набор зависимостей может включать сотни и тысячи библиотек, в нем все равно содержится потенциальная проблема из-за JsonLib:

один потребитель JsonLib (приложение) ожидает версии 1.2.0, тогда как другой потребитель (MQLib) ожидает версии 1.1.5. Перед вами пример проблемы *ромбовидных зависимостей*. Термин возник как описание ситуаций, в которых два потребителя общей библиотеки сами являются библиотеками. Допустим, приложение вообще не зависело от JsonLib, зато от нее зависела DbLib. Возникает ситуация, показанная на рис. 12.6; здесь ромбовидная форма выражена более очевидно.

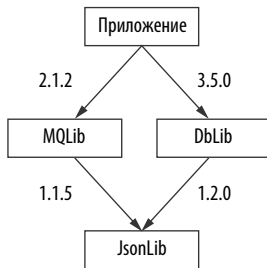


Рис. 12.6. Граф зависимостей с классическими ромбовидными зависимостями

На самом деле на обоих графах присутствуют ромбовидные зависимости. Используемые языки и менеджеры пакетов влияют на то, какие правила применяются к прямым зависимостям от приложения, но главная проблема, которая рассматривается в этом разделе, остается той же.

СЛОЖНОСТЬ ТРАНЗИТИВНЫХ ЗАВИСИМОСТЕЙ

Для простоты мы игнорируем еще один аспект графов зависимостей: как правило, разные версии библиотек имеют разные транзитивные зависимости. Таким образом, JsonLib 1.1.5 может зависеть от CommonLib 1.2.0, а JsonLib 1.2.0 — от CommonLib 1.3.0.

Ключевой вопрос: какая версия JsonLib будет использоваться? И обязательно ли использовать одну версию?

Общие и изолированные зависимости

Разные платформы, языки и менеджеры библиотек/пакетов используют различные подходы к зависимостям. У этих подходов есть свои достоинства и недостатки, и каждый из них имеет свои специфические особенности. Самый важный критерий классификации — являются ли зависимости *общими* для всего приложения или они *изолированы*.

Если зависимость общая, то во всем приложении используется одна версия библиотеки. Если же зависимость изолирована, то каждая из них имеет отдельную копию библиотеки вместе со всем состоянием уровня библиотеки.

Общие зависимости обычно эффективнее изолированных, а иногда они оказываются и более удобными:

- несколько копий кода расходуют больше памяти (и возможно, больше дискового пространства для развертывания) и могут требовать больших затрат на оптимизацию (например, каждая копия байт-кода проходит JIT-компиляцию отдельно);
- одиночные экземпляры ресурсов, требующих затратной инициализации, действуют в границах всего приложения, что повышает его эффективность;
- объекты можно прозрачно передавать между разными компонентами приложения.

Впрочем, также есть два больших недостатка:

- если общее состояние проектируется недостаточно тщательно, компоненты могут вмешиваться в работу друг друга неожиданными способами (например, если каждый из двух компонентов ожидает, что он является единственным пользователем кэша уровня библиотеки, они могут нарушать предположения друг друга);
- если разные компоненты рассчитаны на разные и несовместимые версии одной библиотеки, то единой общей версии, которая подойдет для всех, не существует.

Передача объектов между разными компонентами особенно важна. Допустим, в классической ромбовидной зависимости, показанной на рис. 12.6, три библиотеки содержат следующие классы и методы:

```
public class JsonObject { ... }    ← Класс в JsonLib
public class MQTopic {           ← Класс в MQLib
    public JsonObject readJsonMessage() { ... }
}
public class DbTable {           ← Класс в DbLib
    public void writeJsonValue(string columnName, JsonObject value) { ... }
}
```

Теоретически это очень удобно для приложения: оно может прочитать `JsonObject` из очереди сообщений и записать в базу данных без выполнения преобразований. Это работает, только если типы `JsonObject` в двух сигнатурах методов (возвращаемый тип `readJsonMessage` и второй параметр `writeJsonValue`) на самом деле являются одним типом или, по крайней мере, совместимыми типами.

ПРИМЕЧАНИЕ

В языках со статической типизацией — таких, как Java и C#, — типы с одинаковыми именами, происходящие из разных библиотек, обычно рассматриваются как несовместимые. В языках с динамической типизацией семантика обычно ослабляется. Изолированные зависимости не препятствуют подобной передаче объектов в динамически типизированных языках в той же степени, как в статически типизированных.

Однако это не решает проблему разных несовместимых версий библиотек. Более того, могут возникать дополнительные сложности: например, передача объектов может работать только в одном направлении с одной основной версией, если компонент, создавший объект, использует более раннюю дополнительную версию, чем компонент, потребляющий объект.

Если зависимость одного компонента используется как чисто внутренняя подробность реализации (так, что объекты из нее никогда не возвращаются и не передаются открытому API компонентов), изоляция такой зависимости может быть очень эффективным решением (не считая ситуаций, о которых говорилось выше).

Не обязательно устанавливать все зависимости в приложении только общими или только изолированными. Например, система управления пакетами Maven предлагает возможность создания пакета *fat jar* с изолированным набором всех зависимостей библиотеки. Он может применяться для одной библиотеки, тогда как другие будут использовать общие зависимости.

Проблемы с основными версиями

Предположим, что с общими и изолированными зависимостями мы разобрались, перейдем к рассмотрению последствий от использования несовместимых версий библиотек. Обновим исходный граф зависимостей так, чтобы он требовал несовместимых версий JsonLib, обозначенных разными основными версиями (рис. 12.7).

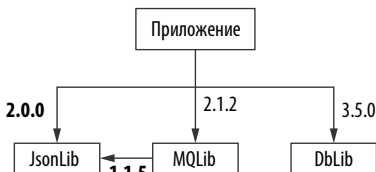


Рис. 12.7. Граф зависимостей с несовместимыми зависимостями

Теперь приложение зависит от JsonLib версии 2.0.0, тогда как MQLib все еще зависит от JsonLib версии 1.1.5. Проблема? Возможно...

Если в приложении используются изолированные зависимости, проблем может не возникнуть. Если приложение написано на языке с динамической типизацией, а объекты JsonLib передаются между приложением и MQLib, это может привести к новым несовместимостям, но в остальном все должно работать.

Если в приложении используются общие зависимости, прежде всего следует понять, какая версия будет использоваться совместно. Возможны три варианта:

- и приложение, и MQLib используют 2.0.0;

- и приложение, и MQLib используют 1.1.5;
- менеджер зависимостей отвергает граф зависимостей как недействительный, так как *ожидаемо* не существует единой совместимой версии.

Самый вероятный вариант — первый. Хорошо ли это? Зависит от того, какие критичные изменения произошли между версиями 1.1.5 и 2.0 библиотеки JsonLib. Вполне возможно, что MQLib не использует нарушенную функциональность; в таком случае все хорошо. Это веский аргумент для менеджеров зависимостей, которые *не слишком* полагаются на графы зависимостей. К сожалению, иногда очень трудно определить, *повлияет* ли критическое изменение на код, особенно если потенциальное нарушение работоспособности находится в другой библиотеке. Даже если инструментарий поможет определить, что с двоичной совместимостью или совместимостью исходного кода все в порядке, он ничего не скажет о семантической совместимости. (Это не умаляет полезности инструментов, которые могут проверить, будет ли комбинация версий библиотек работать при наличии всех необходимых компонентов. Если такие инструменты доступны, мы рекомендуем использовать их. Просто не стоит ждать, что они найдут все возможные нарушения.)

ПРИМЕЧАНИЕ

Различные языки и менеджеры пакетов могут принимать разные решения. Не пожалейте времени и изучите правила и соглашения в контексте, в котором вы работаете и как производитель библиотеки, и как ее потребитель.

Чем сильнее переплетены зависимости приложения, тем выше вероятность, что где-то в них скрывается несогласованность основных версий. И чем больше зависимостей задействовано для одной библиотеки, тем выше вероятность, что такая несогласованность приведет к настоящему нарушению. Это особенно справедливо для самых популярных библиотек — таких, как Apache Commons для Java или Newtonsoft.Json для .NET. (Хотя на момент написания книги у Newtonsoft.Json вышла версия 13.0.1, библиотека имеет очень хорошую историю обратной совместимости.)

Даже если все работает, выход новой основной версии библиотеки требует значительных затрат. Любая другая библиотека, которая зависит от нее и хочет перейти на новую основную версию, должна тщательно проанализировать, не потребует ли это выпуска новой основной версии *с ее стороны*, так как переход может нарушить работоспособность потребителей, зависящих от старой версии. (Влияние версий зависимостей на варианты версионирования кода с зависимостями — отдельная и очень большая тема.) Любое приложение, зависящее от библиотеки, может потребовать внесения изменений в код для перехода на новую версию; возможно, придется решать проблемы ромбовидных зависимостей и рассогласования графов зависимостей.

ПРИМЕЧАНИЕ

Авторы библиотек должны понимать затраты для всей экосистемы и тщательно продумать последствия перед внесением критических изменений.

Я не уговариваю вас отказаться от внесения критических изменений. Проектирование библиотеки API отчасти напоминает порочный круг: часто невозможно убедиться в правильности своих решений или учиться на своих ошибках, пока не появятся пользователи, принявшие вашу библиотеку. На этой стадии исправление ошибок приведет к ошибкам у этих пользователей. Тем не менее тщательное планирование на самой ранней стадии немного повысит шансы на успех.

12.2.3. Снижение последствий от критических изменений

Этот раздел — своего рода «шведский стол» из идей, которые можно выбирать в любом порядке. Общий вывод: нужно тщательно продумывать все, что относится к версионированию.

Знайте контекст: язык, платформу и сообщество

В этой главе я не раз предупреждаю о том, что детали, относящиеся к особенностям языка, важны. Например, вспомните, что переименование параметра не является критическим изменением в Java, но является им в C#. Эти контекстные подробности выходят далеко за рамки того, что считается критическим изменением. Сюда же относится то, как менеджеры пакетов рассматривают графы зависимостей, каковы ожидания сообщества и даже то, какие приемы существуют для обхода критических изменений.

Скажем, *методы по умолчанию* в Java и *реализации интерфейсов по умолчанию* в C# позволяют добавить новые методы в интерфейс без введения критического изменения — по крайней мере, если вы можете предоставить разумную реализацию по умолчанию. (Если разумной реализации по умолчанию не существует, предоставление пустой реализации может создать иллюзию того, что изменение не является критичным, тогда как в действительности оно будет нарушать работу любого кода, вызывающего метод по умолчанию.) Это может повлиять на структуру кода в отношении того, будете ли вы предоставлять интерфейсы или абстрактные классы в библиотеке. Хотя версионирование — очевидно не *единственный* фактор, влияющий на архитектуру библиотеки, важно помнить о последствиях возможных решений для ее будущих версий.

Другой пример, где это важно, встречается в параметрах конструкторов и методов: если вы работаете со списком параметров, который расширяется каждый раз при добавлении новой функциональности, подумайте, можно ли инкапсулировать эти параметры в отдельный тип для обеспечения большей гибкости.

И снова конкретные паттерны проектирования сильно зависят от языка программирования и могут дополнительно ограничиваться целевой платформой. (Например, реализации интерфейсов по умолчанию в C# недоступны в старых версиях .NET.)

Ограничивайте поверхность открытого API

Если вы случайно объявите открытый класс с именем `Costumer` вместо `Customer` и выпустите версию библиотеки с этой опечаткой, ее исправление станет критическим изменением. С другой стороны, если бы этот класс был изначально внутренним для библиотеки, его можно было переименовать в новом исправлении, и пользователь об этом не узнал бы.

Каждый открытый класс, метод, свойство или интерфейс библиотеки становится потенциальной проблемой. С другой стороны, если ничего не объявлять открытым, код невозможно будет использовать, а значит, здесь тоже существует компромисс.

ПРИМЕЧАНИЕ

Часто существует возможность предоставить ограниченную поверхность API изначально — например, с параметрами по умолчанию. После того как пользователи изучат базовую функциональность, вы с большой вероятностью получите полезную обратную связь о том, что требует дополнительной гибкости. Это поможет предотвратить фиксацию библиотеки в конкретной структуре, которая не отвечает потребностям пользователей и не способна развиваться, чтобы удовлетворить их без потери совместимости.

У ограничения поверхности открытого API есть и другой аспект, не столь очевидный, как отказ от объявления открытых классов, которые не нужны пользователям. Ранее было показано, что изменение подробностей реализации вызова одним методом другого метода внутри того же класса может стать критическим изменением при разрешенном наследовании, но быть более гибким, если такие методы не могут переопределяться.

ПРИМЕЧАНИЕ

Иногда из практических соображений приходится раскрывать большую поверхность API, чем того хотелось бы; при этом желательно иметь возможность вносить критические изменения в эти поверхности API. Если четко обозначить, что обычные пользователи не должны соприкасаться с этими типами (например, поместив их в пакет или пространство имен с суффиксом `internal` или вроде того), такое решение становится вполне практичным, даже если оно и не идеально.

Также иногда требуется предоставить доступ к функциональности в стабильном (в целом) выпуске, в котором лишь часть поверхности API все еще нестабильна. В библиотеке Guava именно для этой цели существует аннотация `@Beta`. Решение не идеально (аннотации и имена можно не заметить), но иногда риск оправдан.

Наследование — отличный инструмент, но у него есть свои шероховатости, которые усложняют анализ. Мне нравится рекомендация Джоша Блоха (Josh Bloch): «Проектируйте для наследования или запрещайте его». Там, где вы *проектируете* для наследования и где один переопределяемый метод будет вызывать другой, стоит документировать, что это уже не подробность реализации, — фактически это часть открытого контракта.

Будьте внимательны со своими зависимостями

В предыдущем разделе было показано, как изменения версии могут распространяться по экосистеме. Чем больше общих зависимостей использует библиотека, тем сильнее пользователи будут ощущать изменения в них. Конечно, я вовсе не призываю вас заново изобретать колесо. Прекрасно, если вы можете использовать надежные, хорошо протестированные сторонние компоненты. Просто нужно хорошо представлять последствия от использования этих зависимостей.

Если вы захотите изменить свое решение позже (например, перейти с одной библиотеки разбора JSON на другую), это изменение может быть критическим — и, безусловно, будет, если какие-то типы зависимости используются в открытом API. В инструменте или приложении зависимости обычно изменить проще, чем в библиотеке.

Прежде чем принимать решение об использовании проекта в качестве зависимости в библиотеке, стоит ознакомиться с его историей. Какую часть кода зависимости вы используете? Какова ее политика версионирования? Как быстро разработчики реагируют на отчеты об ошибках и запросы функциональности? Насколько жизнеспособен проект?

Если вы изолируете свои зависимости, каскадный эффект заметно ограничивается, но последствия все равно существуют, так что лучше как следует подумать, прежде чем добавлять новые зависимости. Никто не запрещает вводить зависимости от других библиотек, но делайте это осознанно и с пониманием потенциальных затрат для вас и для потребителей вашей библиотеки.

Решите, что считать критическим изменением

В разделе 12.2.1 вы увидели, что *критичность* не всегда является бинарным атрибутом любого конкретного изменения. Некоторые изменения очевидно критичны для каждого пользователя блока кода; другие нарушают только работу клиентов, использующих библиотеку нестандартным способом.

Когда вы собираетесь внести изменение и хотите оценить, считать ли его критическим, постарайтесь представить наиболее вероятный код потребителя, работоспособность которого будет нарушена этим изменением. Если проблема возникнет, только когда потребитель воспользуется какой-нибудь экзотической возможностью языка, лучше ограничиться увеличением дополнительной версии.

ПРИМЕЧАНИЕ

Казалось бы, можно действовать по принципу «Если сомневаешься, считай изменение критическим». На первый взгляд такой подход кажется осторожным, но в действительности он очень затратен из-за распространения новых основных версий. Время от времени приходится вносить изменения, которые теоретически *очевидно* критичны, но существуют веские аргументы, позволяющие утверждать, что на самом деле у пользователей ничего не сломается. В таком случае лучше всего взять изменение и просто увеличить дополнительную версию. Если вы решите нарушить семантическое версионирование таким образом, не пытайтесь скрыть это — вместо этого открыто и прозрачно изложите причины, по которым вы это делаете.

Одна из «серых зон», которые мы еще не обсуждали в отношении критических изменений, — устаревание функциональности. Во многих языках существует концепция объявления методов или классов устаревшими, что обычно приводит к выдаче предупреждений. Можно ли считать, что вывод нового предупреждения станет критическим для потребителя? А если у него в системе сборки включен режим *интерпретации предупреждений как ошибок*? Лично я считаю это активным решением потребителя: он хочет, чтобы его оповещали о критических изменениях, и мы просто оповещаем его заранее. Как вы вскоре увидите, устаревание может стать мощным инструментом, упрощающим переход пользователей на новые версии.

Все подобные ситуации требуют субъективных решений. Хотя инструменты в некоторых случаях могут уведомлять о критических изменениях, они содержат уже встроенные субъективные решения — например, *добавление нового класса* считается некритическим, хотя оно и *могло бы* создать конфликт имен. (И крайне маловероятно, что он сможет навести на какие-то соображения относительно семантических критических изменений.) По возможности постарайтесь документировать, какие изменения в *вашей* библиотеке считаются критическими, — тем самым вы избавите потребителей от неожиданностей.

Будьте внимательны при повышении основных версий

Наконец, как работать с критическим изменением, когда оно рано или поздно произойдет? Прежде всего я рекомендую сразу вести документ с перечнем всех критических изменений, которые вы собираетесь внести. Каждая основная версия обходится дорого, поэтому изменения стоит группировать, чтобы не нарушать работу пользователей чаще необходимого. И снова не существует единственно верных правил относительно того, с какой частотой стоит выпускать новые версии — это зависит от самой библиотеки и пользователей. Чем больше пользователей и чем больше библиотек зависит от вашей, тем больше будет сложностей с переходом на новую версию.

Постарайтесь как можно яснее выразить свои намерения в документации. В идеале у вас уже есть документ с историей версий, но это особенно важно для

изменений в основной версии. Документируйте каждое критическое изменение, о котором вам известно, а в идеале напишите руководство по миграции в помощь пользователям.

Говоря о миграции, в некоторых случаях можно упростить переход на новую версию за счет использования дополнительной версии как связующего звена. В качестве конкретного примера воспользуемся NodaTime. В NodaTime от 1.0 до 1.3 интерфейс `IClock` объявлялся, как показано в следующем листинге.

Листинг 12.9. Интерфейс `IClock` в NodaTime 1.0-1.3

```
public interface IClock
{
    Instant Now { get; }
}
```

Это было ошибкой; метод слишком похож на `DateTime.Now`, возвращающий системное время, и он не должен быть реализован в форме свойства. Недостаток был исправлен в NodaTime 2.0, как показано в следующем листинге.

Листинг 12.10. Интерфейс `IClock` в NodaTime 2.0

```
public interface IClock
{
    Instant GetCurrentInstant();
}
```

Если бы мы остановились на этом, то пользователь не узнал бы, что было не так или как решить проблему. Вместо этого вскоре *после* выпуска 2.0.0 мы выпустили версию 1.4.0, в которой свойство `Now` было объявлено устаревшим, но был добавлен метод расширения, как показано в следующем листинге.

Листинг 12.11. Упрощение перехода на новую версию в NodaTime 1.4

```
public interface IClock
{
    [Obsolete("Use the GetCurrentInstant() extension [...]")]
    Instant Now { get; }
}
public static class ClockExtensions
{
    public static Instant GetCurrentInstant(
        this IClock clock)=> clock.Now;
```

← Существующие случаи использования `IClock.Now` помечаются как устаревшие

Метод расширения, который выглядит как метод из `IClock 2.0`

Версия 1.4.0 обладала полной совместимостью исходного кода и двоичной совместимостью с 1.3.0, не считая предупреждения. Пользователь мог без проблем игнорировать предупреждения или начать преобразовывать свой код, чтобы подготовить его к переходу на 2.0.0. Для некоторых изменений такой подход неприменим, но в большинстве случаев он работает.

Этот процесс подходит не для всех библиотек, но каждый автор библиотеки может постараться свести к минимуму затраты на критические изменения. Возможно, вы предоставите средства для переноса файлов конфигурации и даже переработки исходного кода. Может быть, это будут средства анализа. А может, у вас нет ничего, кроме документации, но она хорошо написана и в ней разобран практический пример. Умение поставить себя на место другого — чудесный навык: если бы вы использовали эту библиотеку и столкнулись с увеличением основной версии, что бы *вы* хотели увидеть? Последний раздел, посвященный теме версионирования библиотек, рассматривает ее под несколько иным углом, но в нем описывается сценарий, важный для многих разработчиков: библиотеки, предназначенные только для внутреннего пользования.

12.2.4. Управление библиотеками только для внутреннего пользования

В каждой компании, в которой я работал, использовались разные практики версионирования внутренних библиотек. Даже термин «*внутренние*» имеет разный смысл для разных людей: если ваш продукт представляет собой приложение, разбитое на несколько библиотек, но вы не ожидаете, что их будут использовать внешние пользователи, а вся система всегда обновляется за один раз, будут ли эти библиотеки внутренними? Почти наверняка они не должны подчиняться тем же правилам, что и обычные библиотеки.

Аналогичным образом у вас могут быть полноценные внутренние библиотеки, в которых двоичные файлы никогда не попадают на клиентские компьютеры — они просто питают ваш веб-сайт или сетевой API. Нужны ли этим внутренним библиотекам *версии* как таковые? Какие правила действуют при внесении критических изменений в такие библиотеки?

ПРИМЕЧАНИЕ

Вопросы о том, нарушит ли что-нибудь то или иное изменение, становятся более конкретными, если найти весь код, в котором оно используется. В кодовых базах, состоящих из десятков миллионов строк, сделать это довольно сложно — или вообще нереально. Тем не менее эта ситуация гораздо лучше, чем в библиотеках с открытым кодом; часто невозможно определить, как они используются.

Даже если вносить критические изменения относительно легко — например, оставив заметку «Когда команда, ответственная за платежи, переходит на следующую версию библиотеки, она должна изменить свой код», — я рекомендую оценивать свои действия. Старайтесь развивать код потребления вместе с кодом библиотеки, чтобы все продолжало работать, но удаляемые классы (или классы, работоспособность которых будет нарушена) постепенно использовались все реже, так что со временем вы удаляли их без последствий. Даже при этом я рекомендую перед удалением чего-либо вводить своего рода *переходный период*, чтобы при необходимости было проще отменить последние изменения.

Иногда такой постепенный подход не работает или требует слишком больших усилий. Тогда приходится действовать так же, как при миграции схем данных, когда в некоторых случаях короткий простой в работе предпочтительнее риска миграции без отключения и затрат на нее. Впрочем, все зависит от контекста: в некоторых системах есть окна для периодического технического обслуживания, а другие чрезвычайно чувствительны даже к малым простоям.

С уверенностью можно утверждать одно: вносить любые изменения намного сложнее, если у ваших внутренних систем нет четкой стратегии версионирования. Возможно, все компоненты строятся с использованием последних версий всех остальных компонентов. Или это модули с независимым версионированием во внутреннем менеджере пакетов. Может быть, используется гибридный подход с назначением базовых версий одним модулям и стандартными средствами управления исходным кодом для других компонентов. Какую бы схему вы ни выбрали, все члены команды должны понимать систему и то, как изменения в вашем коде могут повлиять на работу коллег.

ПРИМЕЧАНИЕ

Как отмечалось в начале этого раздела, в разных компаниях применяются совершенно разные практики. Иногда команды работают максимально независимо друг от друга — теоретически даже с независимыми системами управления исходным кодом и ограничениями видимости. Такой подход, безусловно, влияет на то, насколько просто организовать безопасную эволюцию внутренних систем с критическими изменениями; при этом он более реален, чем в полностью общедоступной системе. Не жалейте времени на анализ (а затем документирование) процессов, которые могут использоваться для внесения подобных изменений, не нарушая работу других команд или инженерные практики компании.

А сейчас резко сменим курс: вместо *множества библиотек, работающих в одном приложении*, источником сложности станет *множество клиентов, обращающихся с вызовами к одному сетевому API*. У этих двух источников определенно есть общие черты, но они требуют разных подходов.

12.3. ВЕРСИОНИРОВАНИЕ ДЛЯ СЕТЕВЫХ API

Прежде чем переходить к версионированию сетевых API, нужно определить, что иметь в виду под *сетевым API*. Хотя толкований этого понятия существует много, мы примем вариант «сервис „запрос-ответ“, доступный по сети». Возможны и другие (например, API веб-перехватчиков, в которых сервис выдает запрос к пользовательскому коду, а не наоборот), но проще ограничиться случаем, в котором пользователь выдает запрос, а сервис выдает ответ. Лично у меня больше всего опыта работы с простыми сервисами HTTP и данными в формате JSON и сервисами gRPC с использованием Protocol Buffers, но вопросы, которые следует себе задать, применимы ко всему спектру сервисов. (Protocol Buffers — двоичный

формат сериализации Google, который изначально был внутренним, но в 2008 году стал общедоступным. Эта тема более подробно рассматривается в разделе 12.4.) Ответы могут быть очень разными, поэтому стоит помнить о смещении, которое может возникнуть из-за повторного использования ответов из другого контекста.

12.3.1. Контекст вызовов сетевых API

При публикации библиотеки информация о том, как она используется, обычно почти отсутствует (кроме той, что пользователи оставляют в отчетах об ошибках, запросах функциональности, вопросах на Stack Overflow и т. д.). Ожидается, что библиотека предназначена для использования в одной экосистеме — например, у меня никто не спрашивал о том, как библиотека NodaTime взаимодействует с Perl. Экосистема может быть достаточно простой и разнообразной, охватывать несколько языков, но, скорее всего, особых неожиданностей быть не должно.

С сетевыми API (по крайней мере с теми, которые размещаем мы сами) доступно намного больше информации в отношении их использования, поскольку видны входящие запросы. Но при этом обычно неясен контекст, в котором совершается этот вызов API. Такая гибкость становится одним из самых мощных преимуществ сетевых API, но она также усложняет анализ последствий изменений.

Разнообразие контекстов приложений и устройств, отправляющих запросы, не исчерпывается тем, что представлено на рис. 12.8. Можно получать запросы от приложений, написанных на разных языках, — одни строят запросы вручную, другие пользуются специализированными клиентскими библиотеками. Для одной платформы могут существовать разные клиентские библиотеки.

ПРЕДПОЛОЖЕНИЯ О КЛИЕНТСКИХ БИБЛИОТЕКАХ

Клиентские библиотеки могут значительно упростить жизнь пользователей, но их трудно грамотно реализовать, особенно если требуется поддерживать несколько языков и разные API. При превышении определенного размера некоторые части библиотек, по крайней мере большие, будут генерироваться автоматически, либо с существующими инструментами для формата описания API (такими, как OpenAPI), либо с собственным генератором кода. Это создает дополнительную сложность в отношении совместимости: может потребоваться внести в API некоторые изменения, которые совместимы в отношении запросов и ответов, но могут генерировать несовместимые новые библиотеки. Вы сами решаете, допустимо это (с выходом новой основной версии библиотек) или нет.

Даже если вы предоставляете клиентские библиотеки, не стоит предполагать, что все запросы будут генерироваться клиентскими библиотеками (если только вам не удастся обеспечить это, например, при помощи библиотеки, предоставляющей криптографические подписи). Многим API это не нужно, и обычно они менее удобны по сравнению с API, которые легко изменять при помощи таких инструментов, как Postman (<https://www.postman.com/>).

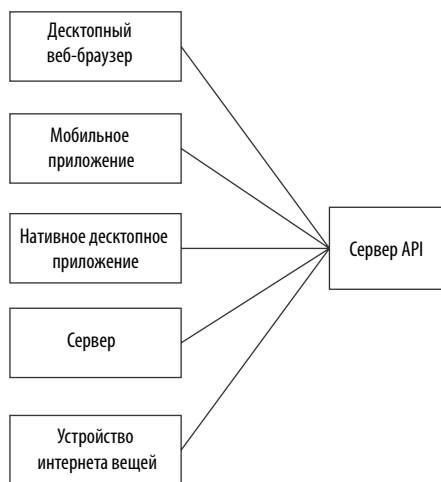


Рис. 12.8. API могут вызываться многими разными устройствами

При обсуждении библиотек и графов зависимостей мы рассматривали приложения, зависящие от библиотек, которые, в свою очередь, зависели от разных версий общей зависимости. У сетевых API прямого аналога не существует, но есть схожие проблемы, возникающие при пересечении старых и новых версий:

- при развертывании старые и новые серверы будут работать в одно и то же время. Современные сервисы обычно необходимо проектировать с таким расчетом, чтобы обновление серверов до новой версии происходило без простоев;
- к одним данным могут обращаться два клиента, один из которых знает только о старой версии, а другой — только о новой версии.

Мы вскоре более подробно коснемся последствий этих проблем, а пока возьмите за правило рассматривать широкий спектр клиентов и условно-согласованный набор серверов. Но сначала разберем цели построения библиотек с точки зрения клиента.

12.3.2. Ясность для клиента

Как это часто бывает, когда вы начинаете размышлять о версионировании для API, возникает искушение сходу погрузиться в технические решения и стратегии. Но без явного заблаговременного определения требований получится либо простая, но никому не подходящая стратегия, либо сложная стратегия, которую будет трудно понять. (Еще хуже, когда стратегия сложна, ее трудно понять, она

не отвечает потребностям клиентов и впоследствии ее слишком трудно перестать придерживаться.)

Стоит задать себе следующие вопросы:

- Спроектирован ли API с расчетом на типичное использование в определенном контексте? (Конечно, в общем случае API *доступны* в разных контекстах. Однако вы можете проектировать API, у которых 99 % клиентов работают на других серверах иначе, чем API, у которых 99 % клиентов работают на IoT-устройствах, неспособных обновляться и крайне чувствительных к размеру ответов.) Это напоминает закономерности трафика и проявления принципа Парето, описанные в главе 5.
- Имеется ли удобный канал связи, по которому можно оповещать всех пользователей о предстоящих изменениях, требующих внимания?
- Намерены ли вы сотрудничать с клиентами по поводу поверхности API, что может привести к выходу версий API с ослабленными требованиями к стабильности?
- С какой ориентировочной скоростью должен эволюционировать API? Насколько быстро клиенты захотят перейти на новейшую и лучшую версию?
- Как долго планируется поддерживать старые версии, отвечает ли это ожиданиям клиентов?
- Можно ли отслеживать использование API в контексте версий, клиентских библиотек и отдельных элементов (например, RPC)?

На некоторые из этих вопросов стоит ответить, даже когда речь идет о простых автономных библиотеках, но сетевые API требуют более тесного взаимодействия. Например, если пользователь желает продолжать пользоваться библиотекой после того, как поставщик прекратил ее поддержку, скорее всего, он сможет это сделать (на свой риск, конечно). В случае сетевых API отключение поставщиком конечных точек, которые этот API обслуживали, немедленно повлияет на работу пользователя.

ПРИМЕЧАНИЕ

Все перечисленные выше вопросы могут повлиять на выбор стратегии версионирования, но одно остается неизменным: клиентам нужна понятная и исчерпывающая документация о версионировании. Она внушает уверенность и коммерческим структурам организации (в том, что они могут положиться на ваш API), и разработчикам (в том, как им планировать свой клиентский код). К сожалению, написанием такой документации обычно пренебрегают; тем не менее документацию, предназначенную для открытого использования, следует сделать неотъемлемой частью стратегии версионирования.

С учетом сказанного рассмотрим два общих подхода, часто применяемых на практике.

12.3.3. Популярные стратегии версионирования

С течением времени в разных организациях сложились разные подходы к версионированию. Похоже, самый распространенный из них такой: *будем надеяться, что проблема не настолько серьезная, и как-нибудь разберемся по ходу дела*. Определенно, это не лучшая идея.

Существуют две более осмысленные стратегии, которые тоже могут привести к катастрофическим последствиям и с трудом реализуются на должном уровне, но, по крайней мере, имеют неплохие шансы на успех. Я обычно называю их *версионированием под контролем клиента* и *версионированием под контролем сервера*. Эти термины не совсем точны, но ниже они разобраны более подробно.

Впрочем, в обоих случаях клиент в том или ином виде указывает версию при выдаче запроса. Вопрос о том, *как именно* она указывается, почти не влияет на процесс принятия решений. Например, в запросе HTTP версия может быть представлена в следующих формах:

- заголовок;
- параметр запроса;
- часть пути в URL.

С другими протоколами возможны и другие формы. У каждого способа указания версии, который вы предлагаете клиенту, есть свои достоинства и недостатки, но эти подробности выходят за рамки книги. Мы сосредоточимся на том, как данный номер версии влияет на API. Начнем с ситуации, в которой клиент может достаточно точно указать версию API, с которой он готов работать.

Версионирование под контролем клиента

При использовании этой стратегии версия, заданная клиентом, определяет *точную* поверхность API, о которой известно коду. Например:

- клиент не должен указывать поле запроса, отсутствующее в этой версии, даже если поле присутствует в другой версии;
- сервер не должен включать в ответ поле, отсутствующее в версии, запрошенной клиентом;
- сервер не должен изменять ресурсы способом, который требует знания полей, отсутствующих в запрашиваемой версии.

Чтобы примеры были более конкретными, представьте очень простой API для работы с ресурсом `Person`, который в версии 1.0 содержит поля `id` и `name`. (На самом деле пример очень упрощен, но так мы сможем сосредоточиться на версионировании. Более подробное описание проектирования обобщенных API приводится в книге Дж. Дж. Дживокса (J. J. Geewax) «API Design Patterns»

[Manning, 2021]). В версии 1.1 появилось новое поле `occupation`. API содержит методы `CreatePerson` и `UpdatePerson`, у которых в запросе задается ресурс `Person`, а также `GetPerson` — в запросе задается идентификатор запрашиваемого ресурса.)

В табл. 12.1 приведены примеры запросов и ответов с версионированием под контролем клиента. В этих примерах используются только методы `CreatePerson` и `GetPerson`; вскоре мы рассмотрим методы `UpdatePerson`.

Таблица 12.1. Примеры запросов и ответов API с версионированием под контролем клиента

Запрос клиента	Ответ сервера	Примечания
Версия: 1.0 Метод: <code>CreatePerson</code> Тело: <code>id=1, name=»Jane»</code>	OK	
Версия: 1.1 Метод: <code>CreatePerson</code> Тело: <code>id=2, name=»Erik», occupation=»Accountant»</code>	OK	Поле <code>occupation</code> может задаваться в запросе версии 1.1
Версия: 1.0 Метод: <code>GetPerson</code> Тело: <code>id=2</code>	OK <code>id=2,</code> <code>name=»Erik»</code>	Хотя ресурс содержит поле <code>occupation</code> , оно не возвращается в ответе версии 1.0
Версия: 1.0 Метод: <code>CreatePerson</code> Тело: <code>id=3, name=»Kara», occupation=»Engineer»</code>	Bad request	Поле <code>occupation</code> не может задаваться в запросе версии 1.0

Формат номера версии достаточно гибок. Он отделяет основную версию от дополнительной по аналогии с тем, как это делается в семантическом версионировании, но номер исправления обычно не включается, так как различия в исправлениях при семантическом версионировании относятся к реализации (или комментариям), а не к API. Дополнительная версия может быть целым числом, которое регулярно увеличивается (с созданием версий 1.0, 1.1, 1.2 и т. д.). Также имеет смысл создать дату из 8 цифр, которая показывает последовательность версий: 1.20200619, 1.20201201, 1.20210504 и т. д. Версии с датой хуже читаются, но дают полезную информацию без необходимости сверяться с полной историей версий.

СТОИМОСТЬ И ЦЕННОСТЬ НОМЕРОВ ИСПРАВЛЕНИЙ

В некоторых нишевых API, для которых критична абсолютная стабильность, можно включить номера исправлений, чтобы клиент, запрашивающий конкретную версию, неизменно получал одно и то же поведение, даже если оно неправильное. Например, API простых чисел имеет версию 1.2.0, которая ошибочно считает 1 простым числом, и эта проблема исправлена (без изменения поверхности API) в 1.2.1. Клиенты, выбирающие версию 1.2.0, все еще получают неправильный результат. Это значит, что нужно заниматься обслуживанием каждой реализации, а это достаточно сложно. Большинству API такой уровень абсолютной целостности не нужен.

Версионирование под контролем клиента затратно в реализации, так как сервер должен знать обо всех когда-либо опубликованных версиях или, по крайней мере, всех версиях, которые планируется поддерживать. Отключение старой версии нарушит работу всех ее существующих клиентов. Конкретная природа нарушения работоспособности клиентов зависит от того, как передается информация об ошибках — как для запрашиваемых версий, никогда не бывших действительными, так и для версий, которые когда-то были таковыми, но сейчас не поддерживаются. Вопрос о том, как это делать, выходит за рамки главы, но этот фактор необходимо учесть при разработке с самого начала.

Один из недостатков версионирования на стороне клиента заключается в том, что реализация должна хранить подробную информацию обо всех дополнительных версиях, чтобы знать, как проверить запрос и какие поля включить в ответ. При распространении запроса в системе также должен распространяться номер версии, заданный клиентом. Стоит автоматизировать процесс проверки запросов и удаления полей, которые изначально не должны присутствовать в версии, заданной клиентом.

Теоретически версионирование под контролем клиента позволяет быстро развивать API без нарушения работоспособности клиентов. Например, если вы допустили опечатку в имени поля в версии 1.0, можно просто запустить версию 2.0 с исправленной опечаткой и преобразовать оба запроса во внутренний формат, который затем обрабатывается без привязки к версии. После этого внутренний формат запроса может быть преобразован обратно в формат ответа для конкретной версии. Хотя такое решение подойдет существующим пользователям версии 1.0, пока они используют 1.0, оно потребует затрат при обновлении до версии 2.0 в отношении внесения изменений в код.

ПРИМЕЧАНИЕ

Основной номер версии в действительности предназначен только для людей; так как номера дополнительных версий фактически независимы, ни коду сервера, ни коду клиента не нужно беспокоиться о том, что переход с 1.0 на 1.1 — это изменение с обратной совместимостью, а переход с 1.1 на 2.0 — это критическое изменение. Для людей важен момент обновления их приложений для перехода на новую версию — они знают, придется ли им обновлять код из-за критических изменений.

У версионирования под контролем клиента есть полезный побочный эффект в отношении циклов «чтение — изменение — запись». Он следует из последнего пункта списка, который вы видели в начале раздела. Вернемся к API с ресурсом `Person`, который в версии 1.0 содержал поля `id` и `name`. В версии 1.1 было добавлено новое поле `occupation`. API может содержать метод `UpdatePerson`, получающий `Person`, который обновляет все поля записи значениями из запроса. Если какое-то поле отсутствует в запросе, то оно очищается.

Если не учитывать поля, о которых известно клиенту, такой подход может иметь рискованные последствия. Рассмотрим упрощенный код для обновления имени, приведенный в следующем листинге.

Листинг 12.12. Простой код для обновления имени

```
public void updateName(String id, String newName) {
    Person person = client.getPerson(id);
    person.setName(newName);
    client.updatePerson(person);
}
```

Код выглядит вполне безобидно. Но что, если клиент знает только о версии 1.0, а у обновляемой записи поле `occupation` задано другим клиентом? В табл. 12.2 содержится последовательность запросов, которая приводит к потере информации в этом API, если на сервере версионирование реализовано неправильно.

Таблица 12.2. Неправильная реализация цикла «чтение — изменение — запись» приводит к потере данных

Запрос клиента	Ответ сервера	Примечания
Версия: 1.1 Метод: CreatePerson Тело: id=2, name=»Erik», occupation=»Accountant»	OK	
Версия: 1.0 Метод: GetPerson Тело: id=2	OK id=2, name=»Erik»	Поле <code>occupation</code> не возвращается, потому что клиенту 1.0 о нем не известно
Версия: 1.0 Метод: UpdatePerson Тело: id=2, name=»Eric»	OK	Клиент 1.0 предоставляет все поля, о которых ему известно
Версия: 1.1 Метод: GetPerson Тело: id=2	OK id=2, name=»Eric»	Значение <code>occupation</code> потеряно

В этом примере виноват сервер с некорректной обработкой в методе `UpdatePerson`. Хотя метод ожидает получить полный ресурс, он может быть полным только с точки зрения того, что понятно клиенту. Тот факт, что клиент не указал значение `occupation`, не значит, что он хочет удалить существующие данные; в этом случае он означает, что клиенту неизвестна концепция `occupation` в ресурсе `Person`.

К счастью, сервер может действовать умнее. Он может учесть тот факт, что клиент указал версию 1.0, и определить, что обновлять следует только поля, присутствующие в этой версии. Это само по себе может привести к непростым решениям при добавлении новых полей, которые должны проверяться по существующим полям, но во многих случаях такой подход достаточно хорош. С грамотно реализованным сервером клиент может выдавать запросы на обновление, не беспокоясь о том, что они снесут какие-то данные, о которых он даже не подозревает. При

этом игнорируется необходимость проверки параллелизма — другого аспекта предотвращения потери данных. Но две причины потери данных фактически ортогональны, и параллелизм не связан с версионированием API (скорее он относится к версионированию *ресурсов*). Реализовать эту функциональность на стороне сервера не всегда просто, но обычно это удастся сделать при помощи относительно обобщенных автоматизированных средств.

А теперь перейдем к версионированию под контролем сервера. Оно не дает серверу полной свободы действий, но, безусловно, немного ослабляет контроль.

Версионирование под контролем сервера

При версионировании под контролем сервера концепция дополнительных чисел не применяется. API может эволюционировать с сохранением обратной совместимости только без изменения основной версии, а клиенты должны просто игнорировать любую возвращенную информацию ответа, которая им непонятна.

При версионировании под контролем сервера все равно должен присутствовать основной номер версии, заданный клиентом, — неважно, в URL, IP-адресе или заголовке, но он должен где-то быть. Без *такого* уровня согласования будет невозможно создать критические изменения, не нарушив работоспособность существующих клиентов.

Версионирование под контролем сервера кажется более динамичным и менее точным, чем под контролем клиента. Обычно оно проще реализуется на серверах, потому что им достаточно поддерживать реализации по количеству основных версий вместо того, чтобы следить за поддержкой всех дополнительных версий по отдельности. Описанный ранее подход с адаптированием запросов и ответов во внутренний формат и из него работает и для версионирования под контролем сервера; просто в нем задействовано меньше адаптеров.

Тот факт, что сервер способен вернуть больший объем информации, чем ожидалось, может создать проблемы для некоторых клиентов. Например, если устройство IoT запрашивает информацию о книге, ожидая получить сводку из нескольких сотен байтов, а API дополнительно включает выдержку в виде первой главы, это может переполнить память устройства при обработке ответа. Эта проблема решаема, и существуют паттерны API для ограничения возвращаемой информации, когда клиенту нужны только ее отдельные части; тем не менее об этом следует помнить.

Цикл «чтение — изменение — запись», рассмотренный выше, создает больше проблем при версионировании под контролем сервера, так как сервер не может знать, какой набор полей известен клиенту. Методы обновления, получающие весь ресурс и безусловно копирующие поля, могут легко привести к потере данных. Поэтому предпочтителен выборочный подход, при котором API проектируется для передачи списка обновляемых полей вместе

с их данными. Изменим API из примера и добавим в него метод `PatchPerson`, получающий ресурс и список полей. В табл. 12.3 приведена последовательность событий, похожая на последовательность из табл. 12.2, но с использованием версионирования под контролем сервера. Информация о том, был ли клиент написан для версии 1.0 или 1.1, уже не является частью запроса; собственно, даже конкретной версии 1.1 может не быть — на момент генерирования кода это просто API v1.

Таблица 12.3. Реализация цикла «чтение — изменение — запись» с семантикой выборочного обновления

Запрос клиента	Ответ сервера	Примечания
Версия: 1 (client 1.1) Метод: <code>CreatePerson</code> Тело: <code>id=2, name=»Erik», occupation=»Accountant»</code>	OK	
Версия: 1 Метод: <code>GetPerson</code> Тело: <code>id=2</code>	OK <code>id=2, name=»Erik», occupation=»Accountant»</code>	Поле <code>occupation</code> возвращается, потому что указана только основная версия. Клиент может игнорировать информацию, которая ему непонятна
Версия: 1 Метод: <code>PatchPerson</code> Тело: <code>resource={id=2, name=»Eric»} fields=»name»</code>	OK	Клиент указывает все поля, которые он хочет изменить. Это могут быть все поля, о которых ему известно, или их подмножество
Версия: 1 Метод: <code>GetPerson</code> Тело: <code>id=2</code>	OK <code>id=2, name=»Eric», occupation=»Accountant»</code>	Значение <code>occupation</code> все еще присутствует, потому что изменяются только перечисленные поля

API, использующие версионирование под контролем клиента, также могут предоставлять семантику выборочного обновления по соображениям эффективности. Но они *не обязаны* этого делать для предотвращения потери данных, поскольку у них есть информация о том, какие поля должны быть известны клиенту. В API, использующем версионирование под контролем сервера, семантика выборочного обновления чрезвычайно важна для всех случаев, кроме простейших.

СОХРАНЕНИЕ НЕИЗВЕСТНЫХ ПОЛЕЙ

Некоторые форматы сериализации также могут сохранять неизвестные поля при разборе ответа и воспроизводить ту же информацию, если эти данные ответа используются в другом запросе. Например, такое поведение поддерживается в Protocol Buffers. Тем не менее это решение может быть ненадежным: если данные ответа десериализуются в другую объектную модель, существует вероятность, что неизвестные поля будут потеряны в этой точке. Явное перечисление полей, которые вы хотите изменить, остается более надежным.

Можно использовать любой из описанных подходов к версионированию. Они имеют разные последствия для версионирования клиентских библиотек, документации, реализации на стороне сервера и даже проектирования самих API, как было показано для обновления ресурсов. Вы сами выбираете для себя приемлемую стратегию, причем она может быть и совершенно иной (хотя я советую очень хорошо подумать, прежде чем отходить слишком далеко от обеих представленных схем).

Существуют и дополнительные аспекты, которые влекут за собой почти одинаковые последствия в обеих стратегиях. Они касаются практических вопросов реализации для хранения данных, которыми мы займемся в последнем разделе этой главы.

12.3.4. Дополнительные аспекты версионирования

Полный анализ всех возможных аспектов версионирования сетевых API выходит за рамки этой книги; он занял бы отдельный увесистый том. Тем не менее о некоторых из них следует хотя бы кратко упомянуть — в основном для того, чтобы вы не забывали продумывать их в специфическом контексте своих API.

Предварительное версионирование

Проектирование API — сложная задача. Часто ее недооценивают; в конце концов, поверхность API сама по себе не содержит логики. Казалось бы, все сложности должны быть связаны с реализацией. Хотя это действительно так для некоторых API, проектирование требует технической квалификации и творческих навыков. Вряд ли вы будете заранее знать, как будет использоваться API (и это одна из самых приятных особенностей его разработки), а это означает, что проектирование ведется на основе крайне неполной информации. Добавьте к этому ограничения на итерации API из-за соображений совместимости, и правильный проект начинает казаться чудом. На помощь приходят предварительные версии.

Предоставляя потенциальным пользователям API его раннюю версию — или раннюю версию новой функциональности, добавляемой в существующие API, — можно получить обратную связь до того, как станет слишком поздно менять полученную поверхность API. Хотя вы всегда *можете* выпустить новую основную версию API для устранения любых проблем, это будет раздражать пользователей, которым придется вносить изменения в свой код. На рис. 12.9 показано, как работают предварительные итерации до и после первого стабильного выпуска API.

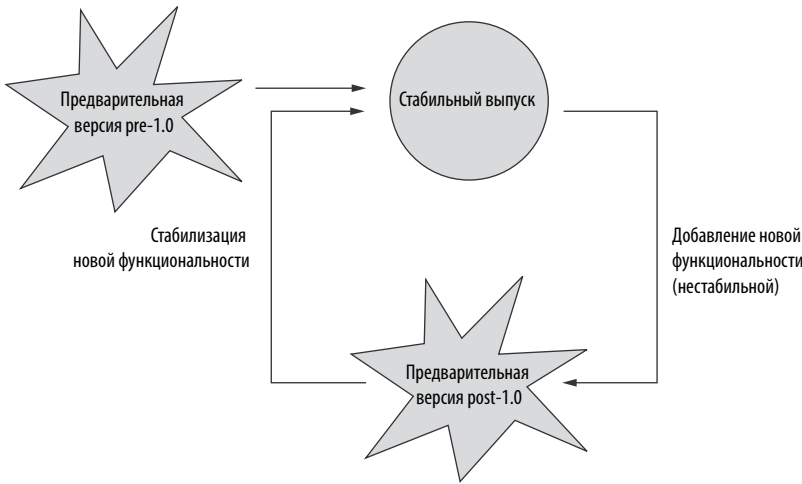


Рис. 12.9. Итерация API с предварительными версиями

В табл. 12.4 показана гипотетическая последовательность выпусков для API. Форма строки с версией не обязательна; все предшествующие обсуждения при сравнении версионирования под контролем клиента и сервера остаются актуальными. Следующая таблица дает представление о том, как пользоваться обратной связью от клиентов для улучшения архитектуры API без нарушения работоспособности клиентов, которым необходима стабильность.

Таблица 12.4. Возможная последовательность выпусков API

Версия	Дата выпуска	Примечания
1.0-alpha.1	10.01.2023	Первый эскиз API для получения обратной связи. Некоторые части не реализованы или очень плохо работают
1.0-beta.1	15.02.2023	Изменения, основанные на обратной связи, и улучшенная реализация — но все еще никаких гарантий стабильности поверхности API или доступности сервиса
1.0-beta.2	25.02.2023	Некоторые критические изменения, основанные на обратной связи от 1.0-beta.1. Может быть заявлена как кандидат на выпуск, если вы чувствуете достаточную уверенность
1.0	05.04.2023	Первый стабильный выпуск с гарантиями стабильности поверхности API и доступности сервиса
1.1-beta.1	08.04.2023	Предварительный выпуск с двумя новыми возможностями (X и Y)
1.1-beta.2	05.05.2023	Обновление, основанное на обратной связи от 1.1-beta.1; возможность X содержит критические изменения поверхности API
1.1	30.05.2023	Стабильная версия, содержащая Y, но не X, так как обратная связь от клиентов сообщает, что X требует доработки

Версия	Дата выпуска	Примечания
1.2-beta.1	30.05.2023	Выпускается одновременно с 1.1, так что клиенты, опробующие возможность X (которая остается нестабильной), могут пользоваться более широкой стабильной поверхностью API, включая Y. Добавляет новую возможность Z
1.3	14.07.2023	Стабильный выпуск, содержащий возможности X и Z. В одновременном выпуске бета-версии нет необходимости из-за отсутствия заметной неустойчивой функциональности. (В больших API это встречается редко. В них почти всегда присутствуют те или иные нестабильные возможности)

Предварительные версии API — не просто обычные версии API, предоставляемые отдельному подмножеству клиентов. Они выпускаются с другими ожиданиями как в отношении поверхности API, так и в плане стабильности и производительности реализации. Важно позаботиться о том, чтобы клиенты разделяли эти ожидания и они ни для кого не стали сюрпризом.

Как и в предыдущих случаях, существуют разные подходы к выпуску предварительных версий. Тем не менее полезно продумать ответы на ряд вопросов.

- Как осуществляется отбор предварительных версий, чтобы клиенты случайно не использовали нестабильную функциональность?
- Должен ли каждый запрос сообщать, что он намерен использовать предварительную функциональность?
- Выполняются ли предварительные версии API полностью независимо от стабильных или один сервер API способен обрабатывать оба вида трафика?
- Какие гарантии предоставляются клиентам относительно стабильности и доступности API (и предоставляются ли)? Например, если в каждом запросе сообщается об использовании предварительной версии API, как долго следует поддерживать работоспособность каждой версии? Будет ли возможность незапланированно внести критические изменения или потребуются короткий переходный период?
- Как предоставить клиентам доступ к предварительным версиям, есть ли для этих версий клиентские библиотеки?
- Как документировать предварительные версии?
- Кто получает доступ к предварительным версиям — все желающие, узкий круг доверенных пользователей или используется промежуточный вариант?
- Понадобятся ли предварительные версии, рассчитанные на конкретные клиенты?
- Какие внутренние инструменты и процессы потребуются для поддержки ответов на все эти вопросы?

Если вы попытаетесь внедрить предварительные версии в стратегию версионирования, при разработке которой они не учитывались, скорее всего, вам придется изрядно потрудиться и у клиентов тоже возникнут проблемы. Даже если вы не собираетесь реализовывать предварительные версии с самого начала, их определенно стоит планировать.

Развертывание сервера

Кому-то это покажется очевидным, но, скорее всего, ваш API будет развернут на нескольких серверах. (Нас бы скорее удивило, если бы это было не так для любого реально эксплуатируемого API.) Это означает, что в любой момент времени эксплуатируется комбинация версий API. Согласно стратегии версионирования и тому, как в запросе выражается версия, предназначенная для конкретного клиента, теоретически *можно* маршрутизировать общие запросы в зависимости от версии. Но обычно проще принять меры, чтобы все серверы обрабатывали все запросы для текущей опубликованной версии, а затем открывать доступ к подробностям обновленного API после завершения развертывания.

Процесс развертывания может выглядеть примерно так:

- развертывание на группе пробных серверов;
- отслеживание ошибок на пробных серверах;
- выполнение тестов для новой функциональности API для этих пробных серверов;
- развертывание на остальных серверах (возможно, за несколько часов и даже дней, в зависимости от размера группы серверов);
- выполнение тестов для новой функциональности API на случайной выборке серверов;
- публикация подробной информации о новых API.

Необходимо быть готовыми выполнить откат в любой точке, а для этого следует продумать, что произойдет с ресурсами, модифицированными с новыми изменениями API в ходе тестирования. Если необходимость в откате возникнет *после* публикации нового API, возможно, придется иметь дело с более широким (и чувствительным) набором ресурсов с новыми заполненными полями, оповещающая клиентов об откате, конечно.

Кроссверсионная обработка ресурсов

Ранее говорилось о применении (хорошо продуманном!) основных версий для внесения критических изменений в API. Многие API работают с долгосрочными ресурсами, и *обычно* клиенты должны иметь возможность использовать разные основные версии для обращения к одним и тем же ресурсам. Конечно,

у этого правила есть исключения: можно отказаться от каких-то ресурсов при переходе от версии 1 (v1) к версии 2 (v2), и логично, что некоторые типы ресурсов могут быть доступны только в более поздних основных версиях. Но многие ресурсы в v1 должны быть доступны в v2 без шлюза (это означает, что после обращения к ресурсу со стороны клиента v2 он становится недоступным для клиентов v1).

Такой подход влияет на способ обращения к ресурсам; сам идентификатор ресурса не должен включать номер версии API. Также он накладывает ограничения в том, что касается проектирования новых основных версий и их реализации. Возможно, вам *хотелось бы* начать «с чистого листа» для v2, но необходимость обслуживать клиентов v1 сильно усложняет полную концептуальную переработку. Нужно понимать, что дело даже не в переходе разработчиков клиентов v1 на v2, хотя этот фактор тоже должен учитываться. На первый план выходят последствия для реализации на стороне сервера. Возможно, у вас небольшая группа клиентов v1, которые не собираются переходить на v2 (так что вам не придется беспокоиться о пути обновления), но, если им вдруг понадобится обратиться к тем же ресурсам в качестве клиентов v2, амбициозные задумки пойдут прахом. Это не значит, что переработка с нуля невозможна; просто затраты на ее реализацию будут существенно выше, чем ожидалось.

Некоторые из этих затрат могут определяться функциональностью используемой системы хранения данных. Так мы подходим к последнему разделу этой главы: проектированию хранилищ данных с необходимой гибкостью в отношении версий.

12.4. ВЕРСИОНИРОВАНИЕ ДЛЯ ХРАНИЛИЩ ДАННЫХ

Мы живем в эпоху больших данных. Несколько десятилетий назад была уверенность, что в основном они будут безопасно храниться в базах данных SQL, и теме эволюции схем SQL были посвящены бесчисленные статьи и главы книг. В этом разделе эволюция данных рассматривается в более общем смысле. В наших примерах используется формат Protocol Buffers, но выводы к нему не привязаны. Существует много других форматов, включая Avro и Thrift, — у каждого есть собственные нюансы версионирования. Этот раздел не пытается заменить документацию по формату, но указывает, на какие ее разделы стоит обратить особое внимание независимо от того, какой формат вы решите использовать. Это относится и к SQL, хотя некоторые решения вполне можно привязать к используемой разновидности SQL.

И хотя этот раздел посвящен хранению данных, многие форматы, о которых пойдет речь, также могут использоваться для сетевых API, а вопросы критических изменений, относящиеся к формату, актуальны и для проектирования

и версионирования API. В предыдущем разделе эти темы намеренно не затрагивались, а материал излагался на более высоком уровне. Но после того, как вы спроектируете высокоуровневую стратегию версионирования API, подробности этого раздела пригодятся для повседневной работы. Начнем с очень краткого знакомства с Protocol Buffers, необходимого для понимания остального материала.

12.4.1. Краткое введение в Protocol Buffers

Protocol Buffers (сокращенно *protobuf*) — формат сериализации, изобретенный и широко применяемый в компании Google, но все чаще используемый в более обширной экосистеме, особенно с *gRPC RPC Framework*. Формат Protocol Buffers разрабатывался прежде всего для эффективного хранения двоичных данных, но он также поддерживает представление JSON.

Файлы схем *protobuf* (*proto*-файлы) обычно используют расширение *.proto*. Их следует хранить в системе управления исходным кодом и относиться к ним так же внимательно, как к другим артефактам исходного кода. *Proto*-файл начинается с настроек, за которыми идет последовательность элементов для определения схемы.

- *Сообщения* — основная часть большинства *proto*-файлов, приблизительный аналог определения типа в большинстве языков программирования. Сообщение состоит из полей и может также включать вложенные сообщения и перечисления.
- *Перечисления* — определение именованных отображений для целых чисел.
- *Сервисы* — используется для определения RPC. Хотя *gRPC* и Protocol Buffers часто используются совместно, вполне возможно спроектировать фреймворк RPC на базе Protocol Buffers без *gRPC* или использовать *gRPC* без данных *protobuf*. В этом разделе сервисы подробно рассматриваться не будут.

Каждое поле в сообщении имеет три основные характеристики:

- *тип* — один из примитивных типов (целые числа, числа с плавающей точкой, байтовые строки или текстовые строки), перечисление или сообщение. Тип также может обозначать *повторяемое* поле, которое, по сути, является списком;
- *имя* — используется в сгенерированном коде и при кодировании сообщений в JSON;
- *номер* — используется в двоичном формате сериализации.

В Protocol Buffers также существуют такие дополнительные элементы, как расширения, наборы *oneof*, отображения (*maps*) и необязательные поля. (*Oneof* — это набор полей; одновременно может быть установлено только одно из них.)

Подробности описаны в документации на странице <https://developers.google.com/protocol-buffers>. Однако это выходит за рамки этой главы, которая посвящена более общим соображениям совместимости.

Как правило, схемы protobuf обрабатываются компилятором protobuf (protoc) для генерирования кода, используемого в библиотеках и приложениях. Хотя теоретически возможно написать код, использующий двоичный формат сериализации напрямую, разработчики очень редко отказываются от использования схем. (В некоторых языках можно написать код для модели данных и снабдить его аннотациями для обозначения номеров и типов полей protobuf.)

Рассмотрим краткий конкретный пример того, как мог бы выглядеть proto-файл в ролевой игре. В данных должен быть представлен персонаж, находящийся под управлением игрока, в том числе имя персонажа, профессия, информация о здоровье и имеющиеся у него предметы (снаряжение). В следующем листинге показано, как могла бы выглядеть proto-схема, чтобы хранить данные.

Листинг 12.13. Пример proto-схемы для представления персонажа игры

```
syntax = "proto3";
message Character {
  string name = 1;
  bytes icon_png = 2;
  Profession profession = 3;
  repeated Item inventory = 4;
  // Максимальное количество слотов
  // снаряжения.
  int32 inventory_slots = 5;
  int32 health = 5;
  int32 max_health = 6;
}
message Item {
  string name = 1;
  // Количество занимаемых слотов в снаряжении.
  int32 slots = 2;
}
enum Profession {
  PROFESSION_UNKNOWN = 0;
  MAGE = 1;
  THIEF = 2;
  WARRIOR = 3;
}
```

Сейчас мы не будем подробно рассматривать эту тему, но используем эту proto-схему для обсуждения потенциальных изменений во времени и их последствий. Еще раз повторю, что этот раздел не был задуман как справочник всех особенностей Protocol Buffers; он всего лишь показывает, о чем необходимо помнить при работе с используемым форматом хранения данных. Для начала разберемся, какие изменения могут создать проблемы.

12.4.2. Что является критическим изменением?

Подобно тому как, согласно закону Хайрама, любое изменение в коде может что-то нарушить, любые обнаружимые изменения в схеме хранения могут создать проблемы, если пользователи работают с данными не очень надежным образом. Но если речь идет о схеме внутреннего хранения данных, любое конкретное изменение может оказаться критическим для некоторого подмножества сценариев, ни один из которых вас, возможно, не интересует. Происходит примерно то же, что с совместимостью исходного кода и двоичной совместимостью, только с гораздо большим разнообразием.

Например:

- Protobuf поддерживает несколько типов для представления 32-разрядных целых чисел со знаком, которые имеют разные форматы сериализации. Переключение типа поля с `int32` на `sint32` изменит смысл хранимых данных, но не изменит API сгенерированного кода.
- Переименование поля `health` в `hit_points` никак не повлияет на хранимые данные, но станет критическим изменением в сгенерированном коде для всех пользователей.
- Генераторы кода Java и C# для protobuf применяют «верблюжий регистр» в именах полей при генерировании методов и свойств. Это означает, что переименование поля `inventory_slots` в `inventorySlots` не повлияет на хранимые данные или сгенерированный код для Java и C#, но отразится на сгенерированном коде в большинстве других языков.
- Добавление значения в перечисление (например, новой профессии `ARCHER`) не вызовет сбоев в сборке или хранении, но весь код, который пытается использовать профессию персонажа, должен либо выполнить конкретное действие с новым значением, либо провести обобщенную обработку (то есть *«Я не знаю смысл этого значения, но на всякий случай сохранию его»*).
- Удаление поля нарушает работоспособность любого кода, который попытается использовать его, но не создает других проблем, даже если поле продолжает присутствовать в хранимых данных.
- Добавление поля не нарушит работу какого-либо кода, даже если используется комбинация старого и нового кода, развернутых одновременно, и старый код читает данные, содержащие новое поле.

Последний пример зависит от реализации обработки неизвестных полей в Protocol Buffers, как упоминалось в контексте ответов API. Вскоре мы рассмотрим эту тему более подробно.

Все приведенные выше утверждения предполагают, что данные хранятся только с использованием двоичного представления protobuf. Если они при этом сохраняются в формате JSON, переименование поля нарушит работоспособность этих

данных: в этом формате номер поля игнорируется, но имя свойства JSON определяется по имени поля. Если вы работаете с форматом данных, имеющим несколько представлений, учитывайте это обстоятельство при внесении любых изменений.

Если используется чисто внутреннее хранилище данных, что позволяет найти и изменить любой код, в котором эти данные используются, может оказаться, что критические изменения в отношении генерирования кода вносить можно и теоретически даже несложно. Очень многое зависит от стратегии версионирования, используемой во внутреннем коде. Внесение изменений, нарушающих формат хранения, — намного более сложная задача, к которой следует отнестись серьезно. Как правило, она решается посредством миграции данных, что требует тщательного планирования. Рассмотрим пример.

12.4.3. Миграция данных в системе хранения

Важно с самого начала усвоить, что существует множество видов миграции данных. Например, бывает, что из одной системы данные переносятся в совершенно иную систему, а иногда они переносятся из одной схемы в другую внутри системы. Рассмотрим случай, в котором изменение вносится в существующую схему, и если бы все делалось за один шаг, это изменение стало бы критическим.

Предположим, необходимо изменить иконку персонажа, добавив возможность менять ее размеры и способы использования. Например, при отображении профиля одного персонажа может показываться крупный значок, а при выводе списка персонажей — мелкий. Сейчас существует только одно поле с именем `icon_png`. И хотя в сообщении `Character` можно добавить новые поля, со временем это усложнит обслуживание и повторное использование логики работы с иконками, если подобное нужно будет проделать для других сущностей (например, предметов или игровых областей). Вместо этого лучше ввести сообщение `IconCollection` для расширения возможностей повторного использования как в схеме, так и в коде.

ПРИМЕЧАНИЕ

Подход с выбором самого простого действенного решения бывает очень полезен при построении прототипа, но может быть рискованным при работе с объектами, которые будет трудно изменить позже, — например, хранилища данных. Невозможно предусмотреть все возможные сценарии, и добавление большей гибкости, чем может когда-нибудь понадобиться, влечет неприятные последствия. Часто при включении примитивного поля в схему стоит, по крайней мере, *проанализировать*, насколько целесообразно вводить новое сообщение, даже если оно содержит всего одно поле.

В итоге сообщение `IconCollection` может стать достаточно сложным, но у нас оно пока будет простым.

Листинг 12.14. Пример proto-схемы для коллекции иконок

```

message IconCollection {
  message Icon {
    bytes data_png = 1;
    int32 width = 2;
    int32 height = 3;
  }
  repeated Icon icons = 1;
}

```

Наша цель — заменить текущее поле `bytes icon_png = 2` в сообщении `Character` новым: `IconCollection icons = 7`. Это нужно сделать, не нарушая работоспособность любых текущих клиентов. Обратите внимание на изменившийся номер поля; это важно для того, чтобы можно было осуществить миграцию данных.

Теперь необходимо выполнить ряд действий для миграции данных:

1. Напишите план с остальными действиями и убедитесь, что он устраивает всех ключевых участников.
2. Добавьте новое сообщение `IconCollection` в схему и поле `icons` в `Character`.
3. Измените весь серверный код, который читает данные из существующего поля `icon_png`:
 - если поле `Character.icons` присутствует, а повторяемое поле в нем содержит как минимум один элемент, используется первый элемент;
 - в противном случае используется прежнее поле `icon_png`.
4. Измените весь серверный код, который записывает данные в существующее поле:
 - присвойте полю `Character.icons` новое сообщение `IconCollection` с одним элементом в повторяемом поле;
 - также присвойте прежнему полю `icon_png` новые данные значка.
5. Разверните новый серверный код.
6. Подождите и убедитесь, что развертывание не придется отменять.
7. Запустите программу миграции, которая проверяет каждое сообщение `Character` в системе. Если поле `icon_png` заполнено, а поле `icons` — нет, скопируйте данные в новую коллекцию `IconCollection` в поле `icons`.
8. Измените весь серверный код и удалите из него все упоминания поля `icon_png`.
9. Разверните новый серверный код.
10. Подождите и убедитесь, что развертывание не придется отменять.
11. Запустите программу миграции, которая проверяет каждого персонажа в системе и очищает поле `icon_png`, если оно заполнено (чтобы не было ненужных устаревших данных).
12. Замените поле `icon_png` строкой `reserved 2` в схеме.

Последний шаг гарантирует, что номер поля 2 не будет случайно использован в дальнейшем. Хотя в норме это не создаст проблем, это дополнительная мера безопасности на случай, если позже обнаружатся неперенесенные старые данные. Они не должны быть ошибочно интерпретированы как что-то другое.

На рис. 12.10 эта последовательность действий изображена в графическом виде. В левом столбце текста описано состояние схемы и данные, сохраняемые на каждом этапе, а в правом — изменения, необходимые для перехода к этому состоянию. Каждый этап должен выполняться очень внимательно и с необходимой паузой, чтобы не приходилось выполнять откат. У вас должен быть план отката на крайний случай, но этого следует по возможности избегать.

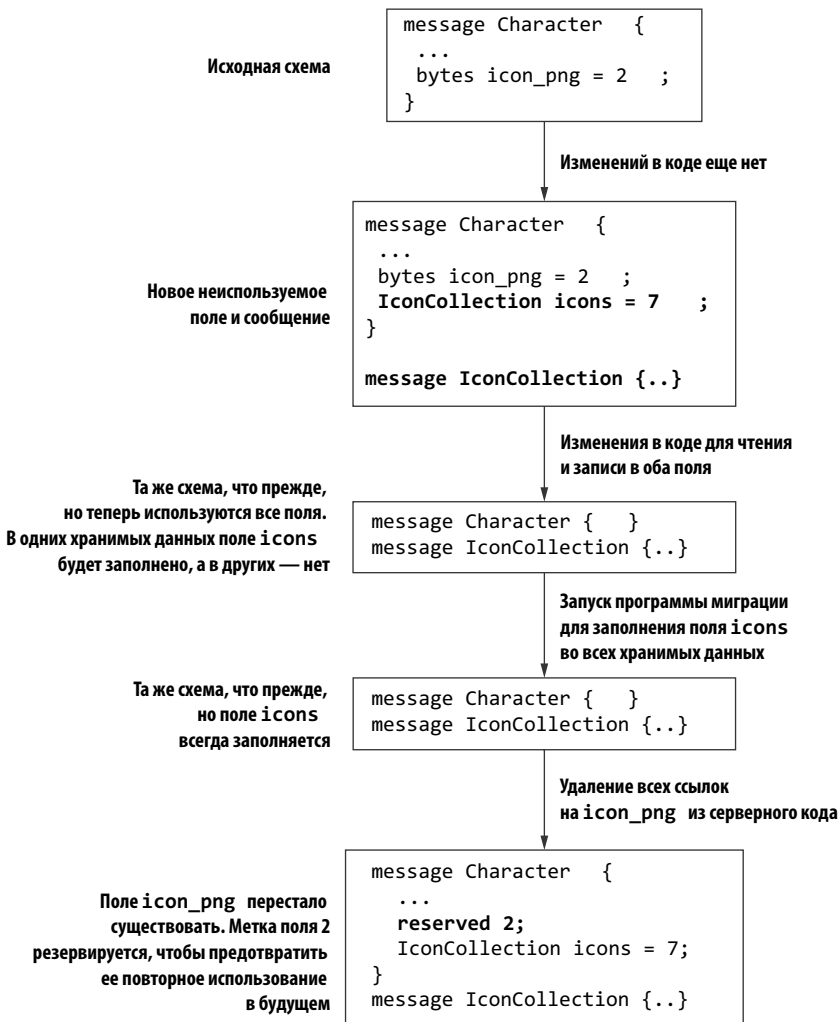


Рис. 12.10. Графическое представление этапов миграции хранилища данных

К этому моменту весь код мигрировал в новое поле, и мы переходим к реализации новой функциональности, что, в свою очередь, может потребовать продуманных шагов в процессе перехода от *одной иконки* к *нескольким иконкам на персонажа*. Этапы ожидания и проверки того, что развертывание не придется отменять, критичны. Такая миграция данных может работать, только когда вы знаете, какой код обращается к данным. При этом неизбежно приходится действовать внимательно, чтобы две версии кода могли параллельно работать с одним хранилищем данных. (Предполагается, что простои абсолютно недопустимы. Если есть возможность полностью вывести систему из эксплуатации для проведения миграции, многое упрощается. Но в современных системах такая возможность обычно отсутствует.) Анализ трех и более разных версий кода, работающих с одними данными, заметно усложняется, и почти всегда лучше просто медленно и стабильно провести процесс. В нашем примере существуют три разные серверные версии:

- исходная версия, которой известно только поле `icon_png`;
- миграционная версия, которой известно об `icon_png` и `icons` и которая гарантирует их согласованность;
- финальная версия, которой известно только поле `icons`.

Если все три версии работают одновременно, то любые изменения в иконке, сохраненные первой версией кода, не будут видны третьей версии кода, и наоборот. Важно и то, что вам известен весь код, который может обращаться к данным, если окажется, что какой-нибудь сервис не получил информацию о новом поле. Это может вызвать серьезные проблемы после теоретического завершения миграции. Именно поэтому самым первым шагом должно стать согласование плана со всеми ключевыми участниками.

Рассмотренные выше шаги совершаются относительно часто, но более сложная миграция может потребовать большего количества шагов, или *громоздких* шагов, — возможно, с миграцией нескольких полей за один раз. Этот процесс всегда сопряжен с некоторым риском и затратами, и при рассмотрении сложной миграции следует учитывать те из них, которые связаны с разбиением миграции на этапы (например, увеличение количества этапов и времени ее выполнения), и сравнивать их с затратами и рисками при ее выполнении за меньшее количество этапов (когда с каждым этапом связан больший риск и он требует более тщательной проверки).

Одно из ключевых предположений в описанной выше процедуре миграции заключается в том, что исходный код может безопасно прочитать сообщение `Character` с полем `icons`, несмотря на то что это поле отсутствовало в схеме на момент развертывания исходного кода. А теперь посмотрим, как это отразится на написании кода.

12.4.4. В ожидании неожиданного

Будем откровенны: никто из нас не силен в предсказании будущего. Мы не хотим, чтобы этот раздел ошибочно считали рекомендацией капитально переусложнить код с учетом любых новых требований, которые могут неожиданно появиться, или рекомендацией зафиксировать требования на ближайшие десять лет, прежде чем вы напишете первую строку кода. Ни одно из этих решений не сработает. Вместо этого я предлагаю проектировать программный код и схемы данных достаточно гибкими. Необходимо, чтобы они могли адаптироваться к будущему без добавления лишней сложности в реализацию текущих требований. Мы рассмотрели один из способов планирования будущего с применением открытых представлений схемы, где мы начинаем с сообщения из одного поля вместо использования примитивного поля.

ПРИМЕЧАНИЕ

Наше сообщение **Character** содержит несколько других примитивных полей. Стоит ли создавать сообщение **Inventory** вместо того, чтобы просто хранить содержимое снаряжения и количество слотов в отдельных полях? А как насчет полей, относящихся к здоровью персонажа? Обычно наличие нескольких полей, связанных с одной обязанностью, по крайней мере наводит на мысль о применении инкапсуляции, как и в программном коде. Однако никаких раз и навсегда установленных правил не существует, и точка принятия решения в схеме хранения данных может отличаться от эквивалентной точки в коде.

Даже такой подход предполагает, что код справится с добавлением новых полей. Современные форматы больших данных обычно проектируются с учетом этого, но необходимо *точно* выяснить, что поддерживается в используемом формате. В частности, в нем могут быть ограничены возможности преобразований в разных представлениях. В **protobuf** *неизвестные поля* (которые будут получены в ходе разбора данных, но были неизвестны на момент генерирования кода по исходной схеме) сохраняются в двоичном представлении, однако затем их невозможно представить в текстовом формате (например, JSON), потому что имена полей не являются частью двоичного формата сериализации.

Операции клонирования, предоставляемые форматом хранения, могут сохранять данные. Но что произойдет, если вручную написать код преобразования одного сообщения схемы в другое? Если смысл части данных неизвестен, очень трудно определить, как они должны участвовать в преобразовании. Каждый раз при написании кода преобразования следует учитывать его влияние на введение новых полей, а при вводе нового поля — помнить о преобразованиях, уже выполняемых с этим сообщением.

Добавление перечислений

Хотя добавление нового поля обычно никак не влияет на существующую логику (при условии, что эта логика может распространять новое значение, не понимая его), с перечислениями ситуация несколько сложнее.

Легко рассматривать перечисление как набор всех известных значений в конкретном контексте, но иногда мы забываем, что «все известные значения» в действительности означают «все известные значения на момент генерирования кода» (или эквивалент этого утверждения). Это все значения, о которых знает код, но это могут быть не все значения, о которых он *когда-нибудь узнает*.

Некоторые перечисления очевидно фиксированы; если вы пишете приложение для обычной карточной игры, будет абсолютно верным создать для представления мастей перечисление **Suit** со значениями **HEARTS**, **CLUBS**, **DIAMONDS**, **SPADES**. С этими значениями удобно работать, и можно написать естественный код, который выдает исключение при передаче значения, не совпадающего ни с одним из вариантов.

Иногда перечисления очевидно предназначены для расширения — как, например, перечисление **Profession** в ролевой игре. При наличии кода, который должен обрабатывать *все* возможные значения **Profession**, необходимо следить, чтобы он обновлялся и разворачивался до того, как вступит в контакт с новым значением, и это стоит отметить при знакомстве с требованиями. Другой код вполне может игнорировать непонятные ему значения перечисления — при условии, что они не будут потеряны. (В Protocol Buffers неизвестные значения перечисления сохраняются в виде числовых значений, а при десериализации и сериализации используется двоичный формат.)

Наконец, некоторые перечисления могут казаться фиксированными, но в итоге выясняется, что они нуждаются в расширении. Например, имеющееся перечисление с названиями всех штатов США остается стабильным очень долгое время, но возможно, в какой-то момент придется реагировать на добавление (и даже удаление) значений. Скорее всего, составлять подробный план на такой случай не нужно, но стоит ненадолго заглянуть в будущее и убедиться, что такое изменение не потребует полной переработки приложения.

В некоторых случаях особенности версионирования перечислений могут вынудить пересмотреть применение перечислений. В частности, там, где для таких значений существует стандартное строковое представление (например, тип MIME или код страны ISO-3166), часто лучше просто сохранить строковое значение.

На нескольких последних страницах мы рассматривали представления данных для хранения, предполагая, что весь код, взаимодействующий с хранимыми данными, полностью контролируется. Рассмотрим это допущение подробнее.

12.4.5. Разделение API и представлений хранения данных

На освоение некоторых популярных приемов работы с данными требуется время, зато они упрощают повседневную жизнь и подходят почти к любой ситуации. Но боюсь, такие приемы не станут ключевой рекомендацией этого раздела. Они чреваты повторением повседневных операций и рутиной (или усложнением инфраструктуры), что обычно раздражает. Но если вы хотите развивать систему, преимущества этих методов перевешивают все недостатки.

Разделяйте схемы для сетевых API и хранения данных

Очень часто мы создаем системы, которые хранят данные, а также имеют сетевой API, используемый для чтения и записи этих данных. Почти в каждой системе с API существуют элементы такого рода, если только они не ограничиваются временными данными наподобие «текущего времени».

После тщательного проектирования схемы хранения данных возникает соблазн также опубликовать ее как схему сетевого API; предполагается, что для обеих схем используется один и тот же формат данных. Храните в точности те данные, которые вы получаете, возвращайте именно то, что было сохранено, — все просто. В некоторых случаях такое решение может стать неплохим первым прототипом, до того как наступит время думать о стабильности. К сожалению, за ускорение на начальном этапе приходится расплачиваться гибкостью в долгосрочной перспективе.

Если сравнить рекомендации этой главы по поводу версионирования сетевых API и хранения данных, оказывается, что они достаточно сильно отличаются, поскольку работают в разных контекстах. С сетевыми AP обычно приходится учитывать, что в любой момент времени найдутся клиенты, использующие схему API в том виде, в котором она была опубликована в разных временных точках, и внесение критических изменений в эту схему потребует огромных затрат. Хотя вы можете убеждать клиентов соответственно изменять свой код, у вас нет возможности влиять на график без чрезмерной навязчивости. (Технически можно предоставить клиентам очень короткое время для перехода на новую основную версию, но, скорее всего, это приведет к потере клиентов, особенно если это происходит слишком часто. *Техническая возможность* и *практическая возможность* — совсем не одно и то же.) В системах с долгим сроком жизни возможность эволюции схемы хранения данных обычно приносит реальную пользу; этот факт стоит признать и встроить его в архитектуру с самого начала.

Конкретный способ разделения двух схем зависит от используемого формата хранения и инструментов, доступных для работы с ним. В своей основе отделение схемы API от схемы хранения данных требует преобразований двух типов:

- преобразование схемы хранения в схему API;
- преобразование данных между двумя схемами.

Обратите внимание, что я намеренно начал со схемы хранения данных: она почти всегда служит источником истины. Между схемой хранения и схемой API всегда существует связь, но ее может быть трудно выразить способом, подходящим для машинного чтения. Это особенно трудно сделать на ранней стадии жизни системы, когда у вас нет времени, которое можно потратить на инструментарий, и вы еще недостаточно хорошо понимаете, какие преобразования вам понадобятся.

Ручные преобразования

Простейший способ *преобразования* схемы хранения данных в схему сетевого API — копирование/вставка/редактирование. Возможно, на первых порах вам даже не потребуется масштабное редактирование, не считая потенциального изменения имени пакета или пространства имен. Изменения в схеме хранения данных можно копировать и вставлять прямо в схему сетевого API — конечно, осознанно.

Преобразование данных во время выполнения (например, преобразование данных из запроса с целью их сохранения или преобразование ресурса, прочитанного из хранилища в ответ) может быть более трудоемким. С технической точки зрения проще всего создать метод для выполнения преобразования во всех направлениях для всех типов схем. Это утомительно, трудоемко и ненадежно. Достаточно легко внести изменение в API хранения и скопировать его в сетевой API, но забыть внести изменение в методы преобразования. Впрочем, это упущение обычно быстро обнаруживается в ходе интеграционных тестов API. (Детализированные интеграционные тесты API жизненно важны по разным причинам, выходящим далеко за рамки версионирования.)

Если все это кажется вам чем-то ужасным и вызывает тошноту, я вас понимаю. Это муторная работа, делать которую не хочется никому. Тем не менее у нее есть свои преимущества. Если вам удастся поддерживать дисциплину *мышления* при внесении каскадных изменений за пределами изменений исходной схемы хранения, вы, скорее всего, обнаружите проблемы или новые возможности. Иногда самое подходящее представление хранения данных для новой функциональности не равно самому подходящему представлению API. Например, применение перечислений в схеме хранения данных имеет свои преимущества, тогда как строки в сетевом API открывают больше возможностей для будущих изменений. Также можно использовать разные уровни детализации и денормализации. Держите в голове различные контексты этих схем и оставьте себе *возможность* принимать разные решения. По мере того как ваша схема растет, а вы обрываете больше уверенности в необходимых преобразованиях, стоит задуматься об автоматизации части работы.

Автоматизированные преобразования

Если ручное ведение разных схем для API и хранения данных становится слишком утомительным, обратитесь к доступным инструментам. Хотя существуют

уже готовые средства, может оказаться, что необходимые инструменты придется писать самостоятельно. Это обеспечивает большую гибкость, но, конечно, увеличивает объем кода, который придется обслуживать.

Я бы воздерживался от автоматизации, не имея достаточного опыта выполнения тех же действий вручную. Такой опыт поможет обнаруживать граничные случаи и аномалии в относительно простой среде с обычным редактированием файлов. Если в дальнейшем вы собираетесь автоматизировать преобразования, обратите особое внимание на граничные случаи, как вы с ними разобрались и почему. Собранный информация упростит процесс автоматизации и предоставит хороший набор тестовых сценариев.

По моему опыту, при разработке средств автоматизации также полезно предусмотреть запасные пути: если конкретный аспект схемы заметно отличается в представлениях API и хранения, может быть проще вручную реализовать эту часть и исключить ее из автоматизации, а не пытаться добавлять новую функциональность в инструментарий (в итоге он будет выполнять все, что нужно, но станет слишком сложным для использования и обслуживания при решении простых задач).

Наконец, что касается преобразования схем, я бы порекомендовал тщательно и вручную проверять вывод инструментария, по крайней мере для нескольких первых изменений схем. А когда инструментарий перестанет преподносить сюрпризы, можно будет исключить ручную проверку из процесса.

Преобразования данных сложнее анализировать, но проще тестировать. И снова инструменты, поставляемые с используемым форматом хранения, могут стать хорошей отправной точкой. Например, библиотеки Protocol Buffers предоставляют API отражения (reflection), который обеспечивает динамический доступ к данным сообщения и хорошую отправную точку для автоматизации преобразований. Места, в которых вам понадобятся запасные пути для преобразований схем, с большой вероятностью потребуют ручного кода для преобразования данных, поэтому, опять же, при проектировании автоматизации стоит обдумать добавление ручного кода, даже если прямо сейчас ничего делать не нужно.

Возможно, вас беспокоит влияние преобразований данных на производительность, особенно после того, как выше я упомянул об *отражении*. Как это обычно бывает с производительностью, стоит оценить необходимые затраты и сравнить эффект от ручных и автоматических преобразований. По моему опыту, затраты общего времени на вызов API редко бывают значительными. Скорее последствия могут ощущаться в отношении затрат памяти и использования процессора; и снова грамотное бенчмарк-тестирование поможет принять взвешенное решение.

Выбор формата хранения данных очень важен, учитывая количество зависящих от него аспектов повседневной работы. Рассмотрим некоторые вопросы, которые стоит задать себе в ходе анализа этого решения.

12.4.6. Оценка форматов хранения

Я не стану рекомендовать конкретный формат хранения или технологию. Решения о выборе системы хранения зависят от многих факторов, в основном никак не связанных с версионированием. Но поскольку эта глава посвящена совместимости, вот некоторые вопросы, которые стоит обдумать при оценке вариантов хранения.

- Поддерживаются ли в вашем варианте те или иные схемы, даже если при этом возможно хранение без схем?
- Предусмотрена ли готовая поддержка эволюции схем? Например, формат Apache Avro с самого начала проектировался с учетом этого фактора и имеет правила совместимости и инструментальные средства для их обеспечения.
- Как обрабатываются непредвиденные значения — например, поля или значения перечислений, которые не присутствовали в схеме, используемой клиентом?
- Как вы собираетесь включать изменение схемы в процесс сборки? Стоит поупражняться в планировании, включая написание последовательности действий для гипотетической миграции данных.
- Если вы планируете использовать сгенерированный код, повлияет ли это на внутреннюю стратегию версионирования? Какую политику вы будете применять к изменениям схемы, которые не нарушают хранение данных, но нарушают работоспособность существующего кода?
- Хотите ли вы использовать один и тот же формат для хранения и представления API? Если да, ответьте на дополнительные вопросы:
 - существуют ли инструменты (или по крайней мере возможность написать собственные) для преобразования схем?
 - существует ли инструмент или поддержка преобразования данных между разными схемами?
 - как эта ситуация вписывается в запланированную стратегию версионирования API?

Стоит отметить, что вопросы из этого списка не подразумевают однозначные ответы («да» или «нет»). Многие технологии хранения данных предоставляют достаточно функциональности для поддержки практически всего, что вы захотите сделать; эти вопросы только помогают оценить, насколько эти технологии упрощают или усложняют ваши задачи. Не забудьте, что при оценке вариантов хранения не стоит пытаться найти лучший формат хранения в мире: выберите то, что лучше всего подходит для вашей системы и вашего контекста.

ИТОГИ

- Суть версионирования — изменение чего-либо со временем. Номера версий содержат важную информацию об этих изменениях в сжатой форме.
- Прямая и обратная совместимость описывает, как новые и старые фрагменты кода и данные могут взаимодействовать друг с другом.
- Семантическое версионирование кодирует информацию совместимости в формате «Основная.Дополнительная.Исправление»:
 - критическое изменение требует новой основной версии;
 - изменение с обратной совместимостью предполагает выпуск новой дополнительной версии;
 - изменение с прямой и обратной совместимостью предполагает новую патч-версию;
 - после номера версии в формате «Основная.Дополнительная.Исправление» может указываться дополнительная информация — предварительный статус, метаданные сборки и т. д.;
 - совместимость в библиотечном коде существует в разных формах. Основные разновидности: совместимость исходного кода (будет ли существующий код строиться с новой версией), двоичная совместимость (будут ли существующие двоичные файлы работать с новой версией) и семантическая совместимость (будет ли существующий код обладать тем же поведением);
 - графы зависимостей приводят к появлению ромбовидных зависимостей, когда разные части одного приложения рассчитаны на разные версии одной зависимости. Критические различия версий зависимости могут сделать невозможным поиск полного набора зависимостей для успешного запуска приложения.
- Основные версии распространяются в экосистеме через графы зависимостей; популярные библиотеки должны вносить критические изменения через новые основные версии как можно реже.
- Внутренний код обычно поглощает критические изменения легче, чем общедоступный, но при этом необходимо действовать осторожно и планировать возможные откаты.
- В общем случае версионирование API сложнее версионирования библиотек; вот самые популярные подходы к нему:
 - версионирование под контролем клиента позволяет ему указать предельно конкретную версию, и ответы никогда не будут включать больше информации;
 - версионирование под контролем сервера позволяет клиенту указать основную версию, а ответы могут включать больше информации, чем понимает клиент.

- Предварительные версии позволяют внедрять пробные изменения, прежде чем принимать их. Пользователю должно быть предельно ясно, что это не стабильная поверхность API.
- Форматы хранения имеют разные характеристики эволюции схемы.
- Сложно проектировать код с учетом всех возможных изменений в схеме хранения данных, но возможность ее изменения следует учитывать с самого начала.
- Отделение схемы API от схемы хранения данных обеспечивает бóльшую гибкость, хотя и приводит к дополнительным затратам — либо из-за рутинной работы, выполняемой вручную, либо из-за автоматизации (с ее потенциальными сложностями).
- Возможно, вам не удастся предсказать все изменения версионирования, которые могут понадобиться, однако время, потраченное на опережающее планирование стратегии версионирования, окупится в долгосрочной перспективе.

13

Современные тенденции разработки и затраты на сопровождение кода

В этой главе

- ✓ Фреймворки внедрения зависимостей.
- ✓ Реактивное программирование и обработка данных.
- ✓ Функциональное программирование в коде.
- ✓ Вычисление отложенное и немедленное.

В сфере разработки ПО регулярно появляются новые библиотеки или концепции (на самом деле практически каждую неделю). Как только вы адаптируете свое приложение или архитектуру к новомодному фреймворку или паттерну, тут же появляется что-то новое. У нас есть микросервисы, реактивное программирование, бессерверные приложения и т. д. У каждого из этих паттернов есть множество преимуществ — таких, как слабая связанность, повышенная производительность или сниженное потребление ресурсов. Тем не менее с каждым из них связаны и определенные сложности.

Для примера предположим, что необходимо перевести все приложение с обработки запросов в отдельных потоках на асинхронный реактивный паттерн. Если решение основано главным образом на современных и популярных тенденциях в программировании, могут возникнуть проблемы. Может оказаться,

что затраты времени и новая модель не подходят для модели обработки приложения.

Прежде чем выбирать новый фреймворк или паттерн, обещающий решить многие проблемы, сначала нужно понять, существуют ли эти проблемы, и измерить их. Если новый фреймворк, используемый в приложении, решает какую-то непростую задачу, он также где-то скрывает дополнительную сложность. Допустим, главная проблема, решаемая заданным фреймворком, не очень серьезна. В таком случае вы используете инструмент, усложняющий приложение без видимых преимуществ. В связи с этим нужно тщательно анализировать достоинства и недостатки нового подхода и его фреймворка перед тем, как его применять. Может оказаться, что дополнительная сложность и затраты, связанные с переходом на новый фреймворк, не оправданы контекстом.

В этой главе продемонстрированы некоторые хорошо известные и проверенные решения для повышения качества разработки, такие как внедрение зависимостей и реактивное программирование. Мы проанализируем, когда стоит следовать новым тенденциям в области разработки и стоит ли применять их в продуктах. Также вы увидите, когда лучше подождать и выбрать более простое, но менее модное решение. Начнем с проверенного паттерна внедрения зависимостей и фреймворков, которые его реализуют.

13.1. КОГДА ИСПОЛЬЗОВАТЬ ФРЕЙМВОРКИ ВНЕДРЕНИЯ ЗАВИСИМОСТЕЙ

Главная идея, лежащая в основе фреймворка *внедрения зависимостей* (DI, Dependency Injection), достаточно проста. Компоненты — сервисы, уровни доступа к данным или конфигурации — не должны конструировать свои зависимости. Вместо этого все зависимости, необходимые конкретному компоненту, должны внедряться *извне*. Впрочем, понятие «извне» не очень четко определено. Это может быть любая вызывающая сторона, предоставляющая реализацию, и внедрение может осуществляться на любом уровне.

Допустим, имеется метод, который должен выполнить операцию с компонентом А. Можно легко внедрить этот компонент через аргумент метода, как показано в следующем листинге.

Листинг 13.1. Внедрение через аргумент метода

```
public void doProcessing(ComponentA componentA){
    // Обработка
}
```

Вызывающая сторона внедряет компонент. Метод `doProcessing()` ничего не знает о происхождении `ComponentA`, потому что он предоставляется извне, и внутри метода `doProcessing()` не создается его новый экземпляр. Тем не менее это может существенно упростить тестирование такого метода.

Можно передать методу имитацию (альтернативную реализацию) и протестировать все явно. Если `ComponentA` вместо этого создается внутри метода `doProcessing()`, это усложнит тестирование метода. Не получится изменить реализацию для целей тестирования. Например, реализация по умолчанию `ComponentA` может потребовать подключения к работающему API другого сервиса. Если удастся внедрить заглушку этого компонента, вызов реального API можно легко заменить фиктивными данными.

Другое преимущество внедрения заключается в том, что метод `doProcessing()` не должен беспокоиться о жизненном цикле `ComponentA`. Операции создания и удаления компонента выполняются снаружи и могут обрабатываться вызывающей стороной.

Передача зависимостей в аргументах — полезный прием. Однако в объектно-ориентированных языках программирования мы обычно строим более сложные объекты, которые, в свою очередь, используют другие объекты. По этой причине внедрение компонентов при каждом вызове метода — не идеальное решение. Передача компонентов при каждом вызове удлинит код и затруднит его чтение.

Ситуация решается внедрением компонента на более высоком уровне с применением внедрения зависимостей через конструктор. При использовании этого метода вызывающая сторона предоставляет все зависимые компоненты во время создания нового экземпляра объекта. Затем эти компоненты присваиваются полям. Наконец, все методы объекта используют ранее внедренные компоненты по ссылкам на поля.

13.1.1. Самостоятельная реализация внедрения зависимостей

Рассмотрим пример настройки внедрения зависимостей, как показано на рис. 13.1. Предполагается, что приложение состоит из четырех компонентов:

- `DbConfiguration` содержит конфигурацию базы данных;
- `InventoryConfiguration` содержит конфигурацию складских запасов;
- `InventoryDb` — уровень доступа к данным, имеющий зависимость от `DbConfiguration`;
- `InventoryService`, главная точка входа приложения, имеет зависимости от `InventoryDb` и `InventoryConfiguration`.

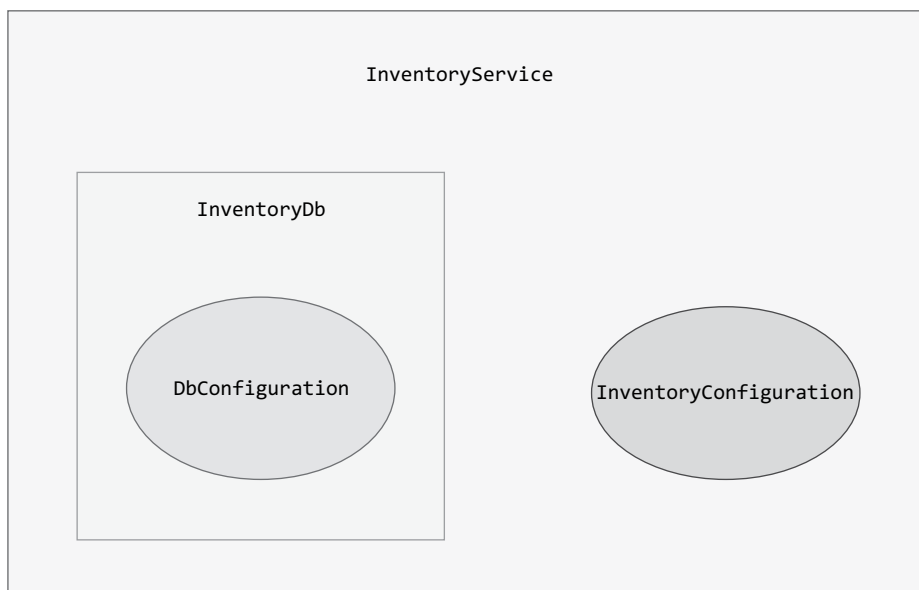


Рис. 13.1. Компоненты паттерна внедрения зависимостей

Поскольку приложение должно строиться с применением паттерна внедрения зависимостей, нельзя создать новый экземпляр одного из этих компонентов в любом другом компоненте. При этом в приложении должно использоваться внедрение зависимостей через конструктор. Это означает, что в приложении должно быть специальное место, в котором сервисы и конфигурации создаются и внедряются, когда это потребуется. А после построения графа зависимостей нужно вызвать метод `prepareInventory()` для `InventoryService`.

В этом сценарии создается класс `Application`, который является точкой входа для приложения. Все зависимости в нем будут создаваться и внедряться по мере необходимости. Процесс создания класса `Application` с внедрением зависимостей показан в следующем листинге.

Листинг 13.2. Самостоятельное внедрение зависимостей

```

public class Application {

    public static void main(String[] args) {
        // Конструирование зависимостей
        DbConfiguration dbConfiguration = loadDbConfig();
        InventoryConfiguration inventoryConfiguration = loadInventoryConfig();
        InventoryDb inventoryDb = new InventoryDb(dbConfiguration);
        InventoryService inventoryService = new InventoryService(inventoryDb,
            inventoryConfiguration);
        inventoryService.prepareInventory();
    }
}

```

Обратите внимание: все сервисы и конфигурации создаются в одном специально выделенном месте. Никакие компоненты не инициализируют другие компоненты внутри себя, что позволяет быстро протестировать любые из этих изменений в изоляции. Например, чтобы протестировать `InventoryService` изолированно, можно внедрить любые объекты `InventoryDb` и `InventoryConfiguration` при конструировании теста. Кроме того, управление жизненным циклом всех компонентов также происходит в одном месте. Например, можно легко закрыть или остановить любой сервис, когда приложение завершит обработку после вызова `prepareInventory()`.

Допустим, необходимо внедрить специализированную реализацию `InventoryDb`. Следующий листинг показывает, как это делается.

Листинг 13.3. Создание специализированного сервиса

```
public class SpecializedInventoryDb extends InventoryDb {
    public SpecializedInventoryDb(DbConfiguration dbConfiguration) {
        super(dbConfiguration);
    }
}
```

После этого в методе `main` можно легко создать другой объект там, где инициализируются все компоненты. Это изменение представлено в следующем листинге.

Листинг 13.4. Изменение инициализации внедрения зависимостей

```
public class Application {

    public static void main(String[] args) {
        // Конструирование зависимостей
        DbConfiguration dbConfiguration = loadDbConfig();
        InventoryConfiguration inventoryConfiguration = loadInventoryConfig();
        InventoryDb inventoryDb =
        ➤ new SpecializedInventoryDb(dbConfiguration); ← Создает новый экземпляр
        InventoryService inventoryService =              SpecializedInventoryDb
        ➤ new InventoryService(inventoryDb, inventoryConfiguration); ←
        inventoryService.prepareInventory();
        }
}
```

Внедряет InventoryDb
в InventoryService

Самостоятельная реализация DI довольно прямолинейна; тем не менее в ней отсутствуют некоторые возможности. Например, представьте, что `InventoryService` не обладает потоковой безопасностью, но в приложении используются несколько потоков. В этом случае можно создать отдельный экземпляр `InventoryService` для каждого потока (или запроса, если это невозможно). Созданное нами решение не предоставляет такой функциональности. По этой причине мы условимся использовать один из фреймворков внедрения зависимостей вместо самостоятельной реализации этого функционала.

13.1.2. Использование фреймворка внедрения зависимостей

Существует целый ряд DI-фреймворков, проверенных на практике, таких как Spring, Dropwizard и Guice. Допустим, мы выбрали фреймворк Spring — он самый популярный и позволяет конструировать сервисы для запросов.

Фреймворк внедрения зависимостей использует DI-контейнер для управления жизненным циклом всех своих компонентов. В Spring такие компоненты называются *bean-компонентами*. Контейнер позволяет регистрировать новые bean-компоненты на стороне их производителя. Он также позволяет получить bean-компонент из контейнера на стороне потребителя. Между производством и потреблением bean-компонентов происходит много событий.

Для bean-компонента можно выбрать разные области действия (<http://mng.bz/0w86>). Он может создаваться один раз за весь срок жизни приложения (паттерн Одиночка). Также возможно создание на уровне запроса, на уровне сеанса и т. д. DI-фреймворк может добавлять дополнительную логику перед вызовом методов bean-компонентов. Например, он способен перехватывать вызовы (с использованием прокси) и регистрировать передаваемые параметры. Поддерживается много разных возможностей.

Переработаем приложение для использования фреймворка Spring. Сначала оба класса конфигурации следует пометить аннотацией `@Configuration` (<http://mng.bz/KBJ0>), как показано в следующем листинге.

Листинг 13.5. Реализация аннотации `@Configuration` в Spring

```
@Configuration
public class DbConfiguration {}

@Configuration
public class InventoryConfiguration {}
```

Затем `InventoryDb` регистрирует себя как аннотацию `@Service` (<http://mng.bz/9Kx1>). Реализация приведена в следующем листинге.

Листинг 13.6. Создание `@Service` в Spring

```
@Service
public class InventoryDb {
    private final DbConfiguration dbConfiguration;

    @Autowired
    public InventoryDb(DbConfiguration dbConfiguration) {
        this.dbConfiguration = dbConfiguration;
    }
}
```

В листинге 13.6 обратите внимание на то, что `InventoryDb` предоставляет как производителя, так и потребителя. В `InventoryDb` также необходимо внедрить `DbConfiguration` перед конструированием его как компонента. Аннотация `@Autowired` сообщает Spring, что перед созданием должен внедряться зависимый компонент. DI-фреймворк обеспечивает порядок инициализации всех его компонентов.

Наконец, `InventoryService` регистрируется как `@Service`. Это определяет область действия компонента как уровень запроса. Каждый раз при поступлении нового запроса создается и внедряется новый экземпляр `InventoryService`. DI-фреймворк также делает это за вас, как показано в следующем листинге.

Листинг 13.7. Определение сервиса с настраиваемой областью действия

```
@Service
@Scope("request")
public class InventoryService {

    private final InventoryDb inventoryDb;
    private final InventoryConfiguration inventoryConfiguration;

    @Autowired
    public InventoryService(InventoryDb inventoryDb, InventoryConfiguration
        inventoryConfiguration) {
        this.inventoryDb = inventoryDb;
        this.inventoryConfiguration = inventoryConfiguration;
    }

    public void prepareInventory() {
        System.out.println("Preparing inventory");
    }
}
```

Посмотрим, как изменился класс `Application` (предыдущая точка входа, в которой инициализировались компоненты). Сначала мы избавляемся от всей логики, относящейся к созданию новых экземпляров. При этом новые экземпляры этих классов все еще можно создавать вручную; тем не менее фреймворк Spring не будет управлять такими экземплярами. Из-за этого возникают два механизма создания компонентов. Ситуация не идеальна; риск ошибок повышается, потому что в приложении мы полагаемся исключительно на Spring DI. Обновленный класс `Application` приведен в следующем листинге.

Листинг 13.8. Измененное приложение со Spring DI

```
@SpringBootApplication
public class Application {
    @Autowired private InventoryService inventoryService;
    public static void main(String[] args) {
```

← Автоматически внедряет
InventoryService

```

    SpringApplication.run(Application.class, args);
}

@PostConstruct
public void useService() {
    inventoryService.prepareInventory();
}
}

```

← Когда все экземпляры будут сконструированы, вызывает prepareInventory()

Метод `main()` изменился по сравнению с предыдущим решением. К тому же мы должны делегировать запуск классу `SpringApplication`, помеченному аннотацией `@SpringBootApplication`. Он сканирует аннотации всех bean-компонентов и обеспечивает внедрение всех необходимых компонентов. Только после того, как все компоненты будут готовы, вызывается итоговый метод `prepareInventory()`.

Здесь стоит сделать несколько важных замечаний. Во-первых, фактическое создание, жизненный цикл и порядок инициализации скрыты. Все это делает DI-фреймворк Spring во внутренней реализации. Пока все работает, нас это не беспокоит. Но продолжая разработку приложения, мы сталкиваемся с проблемами жизненного цикла. Оказывается, что исправлять их намного труднее, потому что вся логика скрыта. Прежде все было под нашим контролем, и мы использовали принадлежащий нам код. Проводить отладку такого кода гораздо проще.

Второй аспект, который нужно учесть, — сильная связанность с фреймворком Spring. Из-за внедрения зависимостей, управляемого аннотациями, наши классы и компоненты *загрязняются* классами фреймворка Spring (или аннотациями). Кроме того, приложение уже не ограничивается простой функцией `main()`. Теперь необходимо делегировать логику запуска фреймворку Spring. Она тоже остается скрытой от нас, и нам придется положиться на этот механизм, если мы хотим использовать внедрение зависимостей.

Наконец, прежде все компоненты логики инициализации находились в одном месте. Теперь инициализация распределяется по кодовой базе. Чтобы представить общую картину жизненного цикла компонентов, придется проанализировать значительный объем кода.

Вернемся к главному аргументу, который заставил нас заменить самостоятельную реализацию на Spring DI. Сначала компонент `InventoryService` был снабжен аннотацией `@Scope("request")` для обеспечения поведения «один сервис на запрос». Однако здесь кроется серьезная проблема. Действительно, новый компонент `InventoryService` будет инициализироваться на уровне запроса при условии, что используется веб-фреймворк, совместимый со Spring DI. На практике это означает, что придется задействовать Spring REST. Появляется еще одна зависимость, и приложение приходится адаптировать для работы с ней. А после реализации первого этапа использования Spring DI приходится делать второй шаг и переводить веб-контроллеры на Spring-совместимый фреймворк.

Чем больше действий мы предпринимаем, тем сильнее привязка приложения к конкретному фреймворку. Кроме того, в приложение импортируется внешняя сложность.

Стоит заметить, что и веб-фреймворк Spring, и Spring DI — проверенные и качественные фреймворки. Но если ваш сценарий использования прост или вы стремитесь ограничить число внешних зависимостей (по разным причинам — см. главу 9), конкретный DI-фреймворк может оказаться не лучшим выбором по сравнению с простой самостоятельной реализацией.

Вместо того чтобы пытаться решить исходную проблему на базе стороннего фреймворка, можно переработать небольшие части собственного решения. Например, построить реализацию `InventoryServiceFactory`, которая создает новый экземпляр реального сервиса при каждом вызове. Ее можно вызывать из той точки, в которой наш веб-сервис получает новый запрос. Даже если все используют конкретный фреймворк, это не значит, что его должны использовать и вы, игнорируя сложность и другие факторы. С другой стороны, если вам нужна проверенная функциональность фреймворка, стоит присмотреться к стороннему решению, даже если у него есть недостатки.

На этом завершается наш анализ внедрения зависимостей как подхода к изменению структуры приложения. Следующий раздел посвящен реактивному программированию.

13.2. КОГДА ПРИМЕНЯЕТСЯ РЕАКТИВНОЕ ПРОГРАММИРОВАНИЕ

Основная идея, лежащая в основе реактивного программирования, — упрощение и повышение эффективности обработки входных данных. Обычно реактивный поток преобразует данные, а затем генерирует результаты. Они могут где-то сохраняться (в некотором приемнике) или потребляться другим кодом, заинтересованным в них. Реактивная модель является неблокирующей; это означает, что обработка должна выполняться асинхронно, а результаты будут генерироваться позже. Кроме того, реактивная обработка должна работать с бесконечным потоком данных и обрабатывать его по мере их поступления (или когда они будут запрошены потребителем).

Реактивное программирование предоставляет механизм функциональной обработки на основе данных, которая происходит без блокирования. Такое программирование обеспечивает высокую степень параллелизма за счет разбиения работы на несколько потоков. При этом модель потоков отделяется от обработки. Невозможно сделать жесткие предположения относительно потоковой модели и того, какой поток будет выполнять те или иные части обработки.

Одна из важнейших характеристик реактивной обработки — поддержка *обратного давления* (back-pressure). Имея поток событий, генерируемых производителем, мы предполагаем, что потребитель может не справиться с одновременной обработкой всех этих событий. В частности, это происходит из-за периодических проблем на стороне потребителя. Если производитель продолжает генерировать события с той же скоростью, а потребитель не может их обработать, события должны где-то буферизоваться. Пока буфер помещается в памяти, проблем нет. Когда потребитель возобновляет процесс с нормальной скоростью, он обрабатывает буферизованные события и возвращается к обычному режиму. К сожалению, если буфер заполнен или в узле произошла ошибка, возникает риск сбоя в обработке и потери части событий.

Для таких случаев реактивная обработка предоставляет механизм, называемый *обратным давлением*. Потребитель сигнализирует о том, что ему требуются новые события для обработки. Производитель получает сигнал и выдает запрашиваемое количество событий. Этот процесс основан на принципе *вытягивания* (pull): потребитель запрашивает события от производителя только тогда, когда он может их обработать. Так формируется естественный механизм обратного давления.

Как видите, реактивная модель предоставляет разнообразные возможности и решает многие сложные проблемы. Тем не менее все это не дается даром. Реактивные API не так просто изучать и анализировать. Они кажутся простыми для стандартных сценариев использования, но для нестандартной обработки они усложняются. Не стоит полагать, что мы нашли решение на все случаи жизни. Чтобы понять это, реализуем конвейер обработки данных и преобразуем его в реактивный.

13.2.1. Создание однопоточной обработки с блокированием

Начнем с реализации процесса обработки, при котором каждый идентификатор пользователя сначала выполняет блокирующий HTTP-запрос GET. Эта операция ввода/вывода выполняется с блокированием и ожидает получения ответа. Второй шаг обработки интенсивно использует процессор и выполняет сложные математические вычисления с числом, возвращенным методом `blockingGet()`. Полученный ответ возвращается вызывающей стороне. Этот сценарий использования изображен на рис. 13.2.

Первая попытка реализации этой обработки будет очень простой. Воспользуемся Java Stream API и выполним эти операции одну за другой. (API, похожие на Java Stream API, существуют и на других платформах — например, LINQ для .NET.) Исходный вариант обработки приведен в следующем листинге.

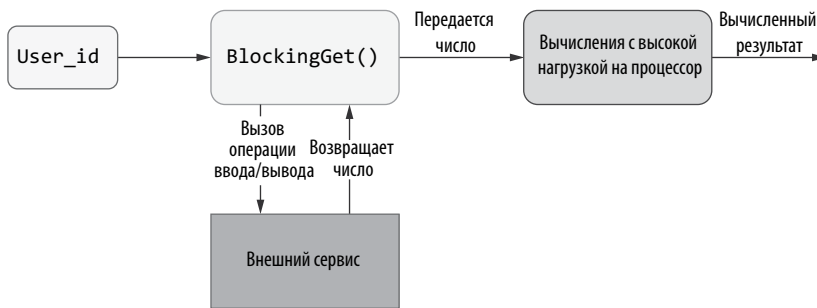


Рис. 13.2. Задачи со вводом/выводом и интенсивными вычислениями

Листинг 13.9. Реализация исходной обработки

```
public List<Integer> calculateForUserIds(List<Integer> userIds) {
    return userIds.stream()
        .map(IOService::blockingGet)
        .map(CPUIntensiveTask::calculate)
        .collect(Collectors.toList());
}
```

После того как все `userIds` будут обработаны, результаты вернутся на сторону вызова. Важно заметить, что реализованная логика является блокирующей. Это значит, что при прямом вызове этого метода (без упаковки его в асинхронное действие) придется ждать, пока метод не завершится. И `IOService`, и `CPUIntensiveTask` регистрируют в журнале поток, в котором выполняются операции (как показано в следующем листинге).

Листинг 13.10. Регистрация потоков в журнале

```
public class CPUIntensiveTask {

    public static Integer calculate(Integer v) {
        System.out.println("CPUIntensiveTask from: " +
            Thread.currentThread().getName());
    }
    // ...
}

public class IOService {

    public static Integer blockingGet(Integer userId) {
        System.out.println("IOService from: " +
            Thread.currentThread().getName());
    }
    // ...
}
```

Напишем модульный тест для этой логики. Сначала генератор `IntStream.rangeClosed` создает список из 10 элементов. Затем все данные передаются

методу `calculateForUserIds()`. Наконец, тест проверяет, что метод возвращает 10 элементов. Код теста приведен в следующем листинге.

Листинг 13.11. Модульное тестирование логики обработки

```
@Test
public void shouldCalculateNElements() {
    // Дано
    CalculationService calculationService = new CalculationService();
    List<Integer> input = IntStream.rangeClosed(1,
        10).boxed().collect(Collectors.toList());
    // Если
    List<Integer> result = calculationService.calculateForUserIds(input);

    // To
    assertThat(result.size()).isEqualTo(10);
}
```

Еще важнее то, что при выполнении этого теста можно наблюдать за потоком, в котором выполняется обработка. Ожидаемый вывод приведен в следующем листинге.

Листинг 13.12. Вывод исходного варианта обработки

```
IOService from: main
CPUIntensiveTask from: main
IOService from: main
CPUIntensiveTask from: main
```

Вся обработка происходит в контексте вызывающего потока. Тест подтверждает, что она выполняется с блокированием и является однопоточной. Таким образом, обработка не параллельна.

13.2.2. Использование `CompletableFuture`

Обе проблемы — блокирование при выполнении и однопоточность — решаются изменением рабочего процесса для использования асинхронной абстракции, доступной в классе Java: `CompletableFuture`. Также существует высокая вероятность, что выбранный язык предоставляет API обещаний/предназначенных значений (futures), позволяющий применить действие без блокирования результатов.

Благодаря этому паттерну можно отправить на выполнение N параллельных задач. Каждая из них может обрабатываться в отдельном потоке или наборе потоков из пула. Мы будем использовать API `CompletableFuture`, встроенный в Java SDK, потому что он уже готов к работе и не требует внешних библиотек.

Посмотрим, как изменится наш метод из предыдущего раздела. Для каждого полученного идентификатора пользователя мы запускаем неблокирующую задачу, которая выполняется в отдельном потоке (отличном от вызывающего потока). Чтобы это сделать, используем метод `supplyAsync()` и вызовем для него первую блокирующую операцию ввода/вывода. Затем присоединим к ней следующую операцию, требующую интенсивных вычислений. Однако эта операция должна вызываться только при завершении первого метода — `blockingGet()`. Для этого воспользуемся методом `thenApply()`, как показано в следующем листинге.

Листинг 13.13. Асинхронная реализация с `CompletableFuture`

```
public List<CompletableFuture<Integer>> calculateForUserIds(List<Integer>
    userIds) {
    return userIds.stream()
        .map(
            v ->
                CompletableFuture.supplyAsync(() -> IOService.blockingGet(v))
                    .thenApply(CPUIntensiveTask::calculate))
        .collect(Collectors.toList());
}
```

Важно отметить, что вычислительная задача выполняется после завершения операции ввода/вывода. Это значит, что нельзя параллельно выполнять два этапа для одного идентификатора. Кроме того, у метода `supplyAsync()` есть разновидность, которой явно передается исполнительный сервис. Это позволяет предоставить собственный пул потоков. Без явной передачи используется стандартный пул ветвления/соединения.

`CalculationService` теперь работает асинхронно и параллельно. Он возвращает список задач `CompletableFuture` с результатами, которые будут видны в будущем. Тем не менее решение о том, следует ли ожидать результата с блокированием или же объединить последующие асинхронные действия в цепочку, остается за вызывающей стороной. Например, она может вызвать метод `get()` для всех операций и объединить результаты в список. Код для тестирования этой возможности приведен в следующем листинге.

Листинг 13.14. Создание теста для асинхронной реализации

```
@Test
public void shouldCalculateNElementsAsync()
    ➤ throws ExecutionException, InterruptedException {
    // Дано
    CalculationService calculationService = new CalculationService();
    List<Integer> input = IntStream.rangeClosed(1,
        10).boxed().collect(Collectors.toList());

    // Если
    List<CompletableFuture<Integer>> resultAsync =
```

```

    calculationService.calculateForUserIds(input);
    List<Integer> result = new ArrayList<>(resultAsync.size());

    for (CompletableFuture<Integer> asyncAction : resultAsync) {
        result.add(asyncAction.get());
    }
    // То
    assertThat(result.size()).isEqualTo(10);
}

```

Ожидает результатов
в блокирующем режиме

Важно заметить, что преобразование между асинхронным и синхронным API может быть достаточно простым. Предположим, новая асинхронная реализация `calculateForUserIds()` вызывается в нескольких местах кода. В таком случае мы не навязываем всем вызывающим сторонам использование асинхронной абстракции `CompletableFuture`. Если код вызывающей стороны работает в блокирующем режиме, он может легко извлечь значения из списка `CompletableFuture` и продолжить использовать API с блокированием. Мы добились параллелизма в своем компоненте, но не обеспечили асинхронный рабочий процесс для всех вызывающих сторон. Запустив тест из листинга 13.14, получим вывод, сходный с представленным в следующем листинге.

Листинг 13.15. Вывод потоков при асинхронной обработке

```

IOService from: ForkJoinPool.commonPool-worker-9
IOService from: ForkJoinPool.commonPool-worker-2
.....
IOService from: ForkJoinPool.commonPool-worker-1
CPUIntensiveTask from: ForkJoinPool.commonPool-worker-2
CPUIntensiveTask from: ForkJoinPool.commonPool-worker-9
...
CPUIntensiveTask from: ForkJoinPool.commonPool-worker-1

```

Обратите внимание: действия выполняются в нескольких потоках. Даже если вызывающая сторона блокирует результаты, реальные вычисления выполняются асинхронно. Допустим, необходимо обеспечить привязку потоков к процессорным ядрам — поведение, при котором задачи с интенсивным вводом/выводом и задачи с интенсивными вычислениями выполняются в одном неглавном потоке. В таком случае методу `supplyAsync()` можно передать однопоточный исполнитель.

Такой подход относительно прямолинеен. В нем используется Java API, доступный для всех потенциальных вызывающих сторон. Можно напрямую влиять на потоковую модель, а ее поведение настраивается достаточно просто. Кроме того, не обязательно, чтобы все вызывающие стороны реализовывали всю обработку асинхронно. `CompletableFuture` без особых проблем упаковывается и преобразуется в блокирующую обработку.

13.2.3. Реализация реактивного решения

Предположим, вы хотите сделать свой код более современным и переработать его для применения реактивной схемы. Обработка выполняет преобразования с N входными элементами, так что реактивная схема кажется подходящей. Решение все еще должно быть асинхронным и параллельным, как в предыдущем варианте. Выбор падает на Reactive API, предоставляющий абстракцию Flux (<http://mng.bz/jyMP>) — реактивный поток из N событий. Другие платформы предоставляют другие библиотеки и фреймворки для реактивного программирования. На веб-сайте <https://reactivex.io/> представлены варианты для разных платформ.

Посмотрим, как работает новая схема. Процесс разбивается на N шагов. Каждое действие из отображения выполняется после завершения предыдущего шага, как показывает код из следующего листинга.

Листинг 13.16. Реализация реактивной схемы

```
public Flux<Integer> calculateForUserIds(List<Integer> userIds) {
    return Flux
        .fromIterable(userIds)
        .map(IOService::blockingGet)
        .map(CPUIntensiveTask::calculate);
}
```

Объект Flux строится по списку элементов методом `fromIterable()`. В реальной реактивной обработке Flux создается из внешних источников и потребляет генерируемые события при их поступлении в систему. Скорее всего, события будут выдаваться постоянно без возможности остановки (горячий источник данных). Реактивный поток — абстракция, позволяющая смоделировать такое поведение.

Как видно из листинга, наш метод возвращает Flux. Его вызывающая сторона должна использовать этот API при взаимодействии с кодом. Возвращение Flux из метода сигнализирует вызывающей стороне, что данные могут выдаваться *бесконечно* (как при стриминговой обработке). Вследствие этого все потребители Flux также должны перевести свой поток на реактивную обработку. Сделать это не так просто, а преобразование элементов Flux в блокирующую абстракцию небезопасно. При использовании потенциально бесконечного производителя данных также возникает риск бесконечного блокирования.

Изменения получаются агрессивными. Внезапно переработка компонента и тот факт, что он использует реактивную обработку, распространяется на все стороны вызова. Реактивная обработка должна быть реализована от производителя до последнего потребителя. Она плохо подходит для распараллеливания небольших частей кода.

Наша цель — создать метод, который не блокирует главный поток, и распараллелить вычисления. Но при запуске нового реактивного кода возникает странное поведение, как видно из следующего листинга.

Листинг 13.17. Вывод для реактивной однопоточной обработки

```
IOService from: main
CPUIntensiveTask from: main
IOService from: main
CPUIntensiveTask from: main
```

Вся обработка выполняется из главного потока! Хотя мы используем реактивный API, обработка остается однопоточной и блокирует вызывающую сторону, потому что она использует главный поток. Как справиться с этой проблемой?

Можно воспользоваться методом `publishOn()` для назначения исполнителя, в котором будет выполняться конкретная часть реактивной обработки. Однако нужно помнить, что метод `blockingGet()` содержит блокирующий вызов ввода/вывода. Согласно спецификации реактивной обработки, действия, выполняемые в реактивном процессе, не должны быть блокирующими. Если это необходимо для вызова блокирующего действия, можно использовать для него конкретный исполнитель `boundedElastic()`, предназначенный для работы с блокирующими вызовами. К сожалению, он показывает не лучшие результаты при выполнении вызовов с интенсивной нагрузкой на процессор, использующих поток в течение значительного времени. Можно воспользоваться исполнителем `parallel()`, оптимизированным для сценария использования с интенсивными вычислениями. Реализация приведена в следующем листинге.

Листинг 13.18. Реактивный параллелизм

```
public Flux<Integer> calculateForUserIds(List<Integer> userIds) {
    return Flux.fromIterable(userIds)
        .publishOn(Schedulers.boundedElastic())
        .map(IOService::blockingGet)
        .publishOn(Schedulers.parallel())
        .map(CPUIntensiveTask::calculate);
}
```

В следующем листинге приведен вывод, полученный при выполнении этого кода. Обратите внимание, что задачи с интенсивным вводом/выводом и интенсивными вычислениями используют разные потоки. Действия чередуются, а это значит, что мы добились некоторого уровня параллелизма.

Листинг 13.19. Вывод потоков при реактивной обработке

```
IOService from: boundedElastic-1
IOService from: boundedElastic-1
CPUIntensiveTask from: parallel-1
IOService from: boundedElastic-1
CPUIntensiveTask from: parallel-1
```

При соблюдении рекомендаций работы с потоками при реактивной обработке трудно добиться привязки потоков к процессорным ядрам. Если вы используете задачи как с интенсивным вводом/выводом, так и с интенсивными вычислениями, они должны выполняться в разных пулах потоков. Следовательно, их обработка в одном потоке нежелательна, в отличие от ситуации с использованием `CompletableFuture` и однопоточных исполнителей.

Мы достигли цели, но у текущего подхода есть недостатки. Во-первых, конфигурация потоков определяется неявно. Можно передать обоим планировщикам уровни параллелизма, но их настройка и оптимизация — непростое дело, которое должно основываться на анализе производительности и тестах. Потоковую модель во `Flux API` тоже не назовешь простой. Предоставляя этот `API` из своих компонентов, мы тем самым требуем, чтобы все пользователи понимали реактивный `API`. Предоставляя доступ к `Flux`, влиять на его использование не получится. Вызывающая сторона может влиять на обработку методом `subscribeOn()` для изменения пула потоков, используемых в коде.

Кроме того, вызывающая сторона не может изменить поток и присоединить следующее блокирующее действие непосредственно для объекта `Flux`, возвращаемого вызовом `calculateForUserIds()`. Такое действие будет выполняться с использованием пула потоков `parallel()`, что повлияет на обработку задач с интенсивными вычислениями.

Конечно, все эти проблемы `API Flux` можно решить, но придется убедиться, что все члены команды понимают реактивный `API`. Если требуется распараллелить только небольшую часть обработки, изменение всего приложения для реактивного `API` может быть слишком агрессивным. С другой стороны, если планируется перестроить всю схему работы приложения на реактивную, проблему следует решать на уровне сквозной переработки (а не одного субкомпонента). Следующий раздел посвящен функциональному программированию.

13.3. КОГДА ПРИМЕНЯЕТСЯ ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Методология функционального программирования имеет много преимуществ: упрощенная модель параллелизма (вследствие неизменяемости состояния), более лаконичный код, упрощенное тестирование (отсутствие побочных эффектов и глобального состояния) и т. д. Тем не менее злоупотребление функциональным программированием в языках, оптимизированных для объектно-ориентированной разработки (таких, как `Java`), может создавать трудности. Рассмотрим некоторые из проблем, возникающих при попытках написания полностью функционального кода в языках, которые в целом считаются объектно-ориентированными.

Язык Java создавался как объектно-ориентированный. К счастью, за последние годы в него были добавлены конструкции функционального программирования, такие как лямбда-функции и Stream API. И хотя эти концепции хорошо известны, они составляют лишь небольшое подмножество конструкций функциональных языков. Возникает соблазн записывать всю логику в функциональном виде просто потому, что эти конструкции доступны. Тем не менее Java по своей сути остается объектно-ориентированным языком.

При попытке создания чисто функционального кода на объектно-ориентированном языке вы столкнетесь со множеством ловушек. Предположим, требуется написать функцию `reduce()` с применением рекурсии. В следующем разделе мы используем для этой цели объектно-ориентированный язык.

13.3.1. Создание функционального кода на нефункциональном языке

Наша цель — написать функцию `reduce()`, которая получает список значений, применяет функцию свертки ко всем этим значениям и возвращает результат вызывающей стороне. Функция должна быть обобщенной, то есть работать для аргументов любого типа.

Также допустим, что вы энтузиаст функционального программирования и хотите реализовать эту логику соответствующим образом. Можно воспользоваться рекурсией и декомпозицией списка. Каждый список можно условно разделить на голову (первый элемент) и хвост (все остальные элементы), как показано на рис. 13.3.

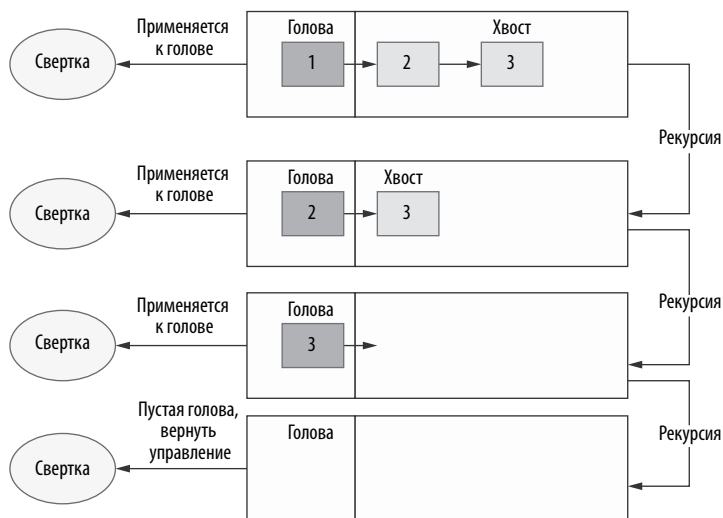


Рис. 13.3. Декомпозиция списка с применением рекурсии

После получения (и удаления) головы из списка можно применить функцию свертки к этому элементу, после чего снова передать хвост той же функции. Снова происходит декомпозиция списка, мы получаем голову, применяем операцию и передаем хвост дальше. Логика повторяется, пока хвост не опустеет. Добравшись до пустого списка, мы возвращаем итоговое значение из рекурсивной функции. Посмотрим, как эта логика реализуется на языке программирования Java с сигатурой метода из следующего списка.

Листинг 13.20. Создание метода `reduce()`

```
public static <T> T reduce(List<T> values, BinaryOperator<T> reducer,
    T accumulator){
    return reduceInternal(values, reducer, accumulator);
}
```

Функция свертки получает текущее накопленное значение в аргументе и голову списка. В первой итерации накопленного значения еще не существует, поэтому вызывающая сторона должна предоставить исходное значение в параметре `accumulator`. Метод `reduce()` делегирует фактическую реализацию методу `reduceInternal()`. Он вызывается рекурсивно, поэтому начать следует с условия завершения, которое указывает, когда функция должна вернуть управление.

В нашем случае значение `accumulator` должно возвращаться при пустом списке `values`. Без этого условия функция никогда не вернет управление и возникнет бесконечная рекурсия. Затем список разделяется на голову (`head`) и хвост (`tail`), как показано в следующем листинге; эта операция делегируется разным методам.

Листинг 13.21. Реализация свертки на Java, `reduceInternal()`

```
private static <T> T reduceInternal
➤ (List<T> values, BinaryOperator<T> reducer, T accumulator) {
    if (values.isEmpty()) {
        return accumulator;
    }
    T head = getHead(values);
    List<T> tail = getTail(values);
    T result = reducer.apply(head, accumulator);
    return reduceInternal(tail, reducer, result);
}
```

После выделения головы можно вызвать функцию свертки, передать ей значения `head` и `accumulator`. Наконец, мы снова вызываем метод (рекурсия).

Методы выделения головы и хвоста достаточно просты. Реализация приведена в следующем листинге.

Листинг 13.22. Выделение головы и хвоста

```
private static <T> List<T> getTail(List<T> values) {
    if (values.size() == 1) {
        return Collections.emptyList();
    }
    return values.subList(1, values.size());
}

private static <T> T getHead(List<T> values) {
    return values.get(0);
}
```

Метод `getTail()` возвращает пустой список, если `values` содержит только один элемент (голову). В противном случае он возвращает все элементы, кроме первого. Метод `getHead()` возвращает первый элемент списка.

Реализуем модульный тест, который проверяет функциональную реализацию `reduce()`, используемую для суммирования всех элементов списка. В следующем листинге приведен код теста.

Листинг 13.23. Создание модульного теста для реализации свертки на Java

```
@Test
public void shouldReduceTenValues() {
    // Дано
    List<Integer> input = IntStream.range(1,
        10).boxed().collect(Collectors.toList());

    // Если
    Integer result = Reduce.reduce(input, (value, accumulator) -> value +
        accumulator, 0);

    // То
    assertThat(result).isEqualTo(45);
}
```

Функция свертки получает значения `head` и `accumulator` и складывает их. Начальное значение `accumulator` равно нулю, потому что сложение начинается с него.

Пока все выглядит привлекательно. Нам удалось реализовать функциональную конструкцию с применением функционального подхода (рекурсии) в нефункциональном языке. Однако у этой реализации есть один серьезный недостаток. Чтобы обнаружить проблему, нужно написать модульный тест, работающий с большим количеством значений. Например, как показано в следующем листинге, при выполнении логики для 100 000 элементов произойдет исключение `StackOverflowError`.

Листинг 13.24. Исключение `StackOverflowError` при тестировании `reduce()`

```

@Test
public void shouldStackOverflowForALotOfValues() {
    // Дано
    List<Integer> input = IntStream.range(1,
        100_000).boxed().collect(Collectors.toList());

    // Если
    assertThatThrownBy(() -> Reduce.reduce(input, Integer::sum, 0))
        .isInstanceOf(StackOverflowError.class);
}

```

Из-за чего выдается `StackOverflowError`? Как выясняется, рекурсия плохо оптимизирована для языка Java. Каждый рекурсивный вызов должен выделить фрейм в стеке вызовов. Для обработки 100 000 элементов это потребует создания равного количества фреймов стека. Каждый такой фрейм занимает некоторую память. Количество элементов в стеке ограничено сверху памятью, доступной для программы. Следовательно, трассировка стека имеет ограниченную глубину. Если код содержит слишком много вызовов, в итоге выдается исключение, которое сообщает о проблеме.

Перед вами один из граничных случаев функционального программирования в нефункциональном языке. Функцию `reduce()` можно реализовать в императивной форме со стандартным циклом `for`, и в объектно-ориентированных языках лучше делать именно так.

Отметим, что метод `reduce()` доступен в Java Stream API, что позволяет безопасно использовать его из Java. Дело в том, что он реализован в императивной форме (цикл `for`), а не в рекурсивной. Мы реализовали собственную функцию `reduce()`, только чтобы показать одну из стандартных проблем функционального программирования (рекурсии) при написании кода на объектно-ориентированном языке, таком как Java.

13.3.2. Хвостовая рекурсия

Если вы работаете на полностью функциональном языке, проблема с рекурсивной реализацией решается легко. Например, в языке Scala реализована такая оптимизация, как хвостовая рекурсия.

Эта оптимизация уровня компилятора направлена на раскрутку рекурсии, что возможно, только если используемый рекурсивный вызов является последним вызовом в методе. В таком случае компилятор преобразует рекурсию в цикл `for`. Вы пишете рекурсивный, полностью функциональный код, не беспокоясь о чрезмерном разрастании стека. Чтобы увидеть, как просто реализовать

функциональный рекурсивный метод `reduce()` в полностью функциональном языке (Scala в данном случае), возьмем реализацию из следующего листинга.

Листинг 13.25. Реализация `reduce()` на Scala

```
@tailrec
def reduce[T] (values: List[T],
  ➤ reducer: (T, T) => T, accumulator:T ): T = values match {
  case Nil => accumulator
  case head :: tail => reduce(tail, reducer, reducer(head, accumulator))
}
```

Код более краток и абсолютно безопасен для выполнения с большим количеством входящих значений. Его компактность обеспечивается еще одной конструкцией функционального программирования: поиском по шаблону с декомпозицией. Декомпозиция списка на голову и хвост реализуется простой конструкцией `head :: tail`. Заметим, что метод `reduce()` помечается аннотацией `@tailrec` (<http://mng.bz/W7J1>). Она приказывает компилятору проверить, можно ли применить к заданному методу хвостовую оптимизацию. Если нет, компилятор возвращает ошибку. Но в данном случае это возможно, потому что рекурсивный вызов является последней командой метода.

Анализируя этот пример и просматривая реализации на Scala и Java, можно сделать вывод, что очень важно правильно выбрать язык и инструменты для конкретной задачи программирования. Функциональное программирование имеет много преимуществ. Но при бездумном применении всех приемов и паттернов возникает риск множества проблем. С одной стороны, необходимо стремиться к адаптации лучших идей из языков функционального программирования. С другой — нужно с осторожностью применять конструкции функционального программирования в языке, который не является чисто функциональным. Используя подходящие паттерны, можно предоставить API, хорошо сочетающийся с принципами функционального программирования (например, `Stream.reduce()`), но основанный на императивной реализации.

13.3.3. Использование неизменяемости

Неизменяемость — мощная концепция, но она имеет свою цену. После создания неизменяемый объект невозможно модифицировать. В языке Java можно создать такой объект, объявив все его поля с `final`. Однако `final`-ссылка означает лишь то, что этой ссылке не может быть присвоено новое значение.

Объект может быть модифицирован, если он был создан способом, допускающим изменение. Создать неизменяемый объект можно, но это требует тщательного проектирования классов. Все способы его модификации должны быть скрыты от вызывающих сторон. Если вы используете конструкцию API, допускающую

изменения (например, `List`), необходимо упаковать изменяемую структуру в неизменяемую обертку. После упаковки остается запретить вызовы любых методов, допускающих изменение упакованного объекта.

Когда объект станет неизменяемым, можно передавать его между компонентами, не беспокоясь о потоковой безопасности. К объекту можно обращаться только для чтения, так что все потоки будут иметь одинаковый уровень видимости. Следовательно, при обращениях к объекту синхронизация не понадобится, что повлияет на производительность кода.

Кроме того, такой код проще писать и анализировать, поэтому при его создании проще избегать багов. Важно, что состояние объекта заполняется во время конструирования — и не позже.

В реальности даже если объект является неизменяемым, иногда требуется изменить его состояние. В функциональной парадигме это делается созданием нового объекта, копированием состояния исходного объекта и изменением того, что требуется. Но после создания копия должна подчиняться тем же правилам, что и исходный объект: правилам неизменяемости. Видим, что такой подход приводит к созданию множества объектов. Каждый из них выделяет часть пространства памяти. Чем больше глубоких копий исходного объекта создается, тем больше памяти для них понадобится. Следовательно, функциональный подход к написанию кода может привести к большим затратам памяти и более дорогостоящей сборке мусора. Необходимо тщательно следить за количеством созданных объектов и их влиянием на сборку мусора.

На практике количество копируемых объектов можно сократить. Для примера возьмем неизменяемую реализацию `List`. Весь список является неизменяемым и состоит из узлов, соединенных указателями. Допустим, ссылка `list1` указывает на список из двух узлов. Вы хотите создать новый список `list2`, который основан на неизменяемом списке `list1`, но содержит один дополнительный узел со значением `C`. Эта реализация изображена на рис. 13.4.

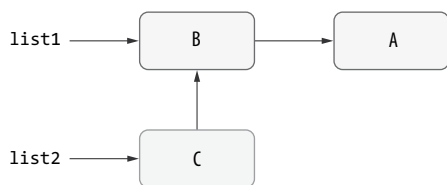


Рис. 13.4. Неизменяемый список

Можно скопировать все узлы из `list1` и добавить новый узел, но такой список будет занимать память для трех дополнительных узлов. Лучше создать один новый узел и вставить в него указатель на голову `list1`. После этой операции

появятся два неизменяемых списка: `list1` с двумя узлами и `list2` с тремя узлами. При этом память расходуется только на три узла из пяти (исходный список `list1` содержит два узла, а новый список `list2` — три узла). Аналогичные паттерны могут использоваться в целях сокращения затрат памяти для других неизменяемых структур и объектов.

Функциональное программирование — сложная тема, требующая более основательного разбора. В этом разделе я лишь хотел продемонстрировать его отдельный аспект и проанализировать его в контексте объектно-ориентированного языка. Если вы хотите больше узнать о функциональном программировании, рекомендую книгу Пьер-Ива Сомона (Pierre-Yves Saumont) «Functional Programming in Java» (Manning, 2017) (<http://mng.bz/8lVw>). В следующем разделе рассматриваются два способа инициализации: отложенный и немедленный.

13.4. ОТЛОЖЕННОЕ И НЕМЕДЛЕННОЕ ВЫЧИСЛЕНИЕ

Приложения обычно взаимодействуют с несколькими компонентами. Рассмотрим веб-приложение, которое должно подключаться к базе данных (с открытием сеанса) и заполнять кэш последних идентификаторов пользователей. В среде с N экземплярами сервиса все эти задачи должны выполняться на каждом узле, как показано на рис. 13.5.

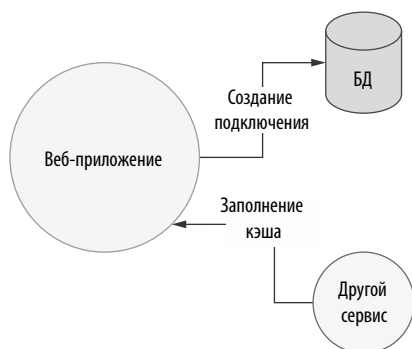


Рис. 13.5. Приложение с двумя используемыми компонентами

Обе операции могут выполняться в отложенном или немедленном режиме. В последнее время существует тенденция к максимальному ускорению запуска приложения. Цель может быть достигнута перемещением долгих операций — таких, как инициализация подключения к базе данных, — на более позднюю стадию жизненного цикла приложения. В этом паттерне подключение к базе данных создается в отложенном режиме. Значит, при запуске приложения подключение не инициализировано. Выполнение логики инициализации откладывается до

первого запроса пользователей. Но это также означает, что первому пользователю придется нести затраты на инициализацию подключения при первом запросе. На рис. 13.6 показано, как может выглядеть отложенная инициализация.



Рис. 13.6. Отложенная инициализация при подключении к базе данных

Немедленная инициализация требует затрат при запуске приложения. В этом сценарии первый пользователь использует уже существующее подключение без необходимости его инициализации. Подключение создается при запуске приложения (и хранится в пуле подключений). Первый запрос получает подключение из пула и использует его для выполнения запроса. На рис. 13.17 показано, как происходит немедленная инициализация.



Рис. 13.7. Немедленная инициализация при подключении к базе данных

В реальных приложениях эффект будет еще более заметным. Часто существуют N подключений к используемой подсистеме и поддерживается пул из N подключений. Пул может расширяться динамически, но он начинается с некоторого стабильного числа открытых подключений. Если вы решите использовать отложенную инициализацию для всех подключений, затраты на нее будут распределены по N запросам, где N равно количеству подключений в пуле.

Придется выбирать между увеличением времени запуска и повышением продолжительности обработки первого запроса (или нескольких начальных запросов). Если предположить, что условия SLA жестко ограничены и не позволяют превысить заданное время обработки запроса, отложенная инициализация может создать проблемы. С другой стороны, если приложение должно запускаться как можно быстрее, перемещение на более позднюю стадию всей логики, требующей больших временных затрат, может быть оправданно.

Немедленная инициализация может создать потенциальную проблему, влияющую на приложение. Допустим, необходимо выбрать момент для заполнения кэша. Можно сделать это сразу при запуске или же выбрать отложенную инициализацию и заполнить кэш во время выполнения запроса. Такое решение очевидно влияет на тот же аспект, что и предыдущий пример: затраты времени будут приходиться на запуск или первые N запросов.

Следует понимать, что при внешнем вызове могут возникнуть проблемы. Например, он может завершиться неудачей из-за недоступности сервиса. Таким образом, есть риск возникновения ситуации, в которой у другого сервиса, используемого для заполнения кэша, появятся проблемы. Также при чтении данных могут возникнуть программные ошибки.

При использовании отложенной инициализации все проблемы обнаружатся во время выполнения приложения. Это может произойти намного позже его запуска. В таком сценарии можно развернуть приложение и прийти к выводу, что все работает, как ожидалось. И только когда оно начнет обслуживать трафик, обнаружатся проблемы. Задержка обусловлена отложенной инициализацией и смещением логики на более позднее время. Если приложение выполняет эти действия немедленно (при запуске), потенциальные проблемы выявляются быстрее.

Если приложение содержит программные ошибки, они могут стать явными сразу же при развертывании нового узла с новой версией. Видя это, можно быстрее отменить развертывание. Это даже может пройти незаметно для конечных пользователей, если использовать плавное (rolling) обновление, при котором старая версия приложения не удаляется, пока новые узлы не начнут работать в нормальном режиме. При отложенном подходе сбой может остаться незамеченным на стадии развертывания. Только после развертывания всех узлов вы (и конечные пользователи) заметите проблему. В табл. 13.1 приведено краткое сравнение отложенной и немедленной инициализации.

Таблица 13.1. Отложенная и немедленная инициализация

Фаза инициализации	Время запуска	Время первых N запросов	Обнаружение ошибок
Отложенная	Быстрее	Влияет; медленнее	Позже, когда сервис начнет работать
Немедленная	Медленнее	Не влияет	Во время разработки

Как видите, отложенная инициализация ускоряет время запуска. При этом затраты времени не исчезают, а распределяются между первыми N запросами к сервису. Кроме того, потенциальные ошибки выявляются позже, когда сервис уже работает.

Немедленная инициализация перемещает затраты времени в фазу запуска. Таким образом, запуск приложения с немедленной инициализацией происходит медленнее, и благодаря этому время обработки первых N запросов не изменяется. Кроме того, некоторые потенциальные ошибки могут обнаруживаться в фазе развертывания.

Решение о том, выполнять инициализацию приложения в отложенном или немедленном режиме, необходимо принимать с учетом этих факторов. Также можно выбрать гибридное решение, при котором одни действия выполняются немедленно, а другие позже.

ИТОГИ

- При внедрении зависимостей те из них, которые необходимы конкретному компоненту, следует внедрять *снаружи*, причем это может происходить на любом уровне. В этой главе вы узнали, когда этот паттерн стоит реализовывать самостоятельно, а когда лучше воспользоваться существующими решениями.
 - Хотя паттерн внедрения зависимостей в методы вполне жизнеспособен, он плохо подходит для объектно-ориентированных языков, таких как Java. Проблема решается внедрением компонентов на более высоком уровне с внедрением зависимостей через конструктор.
 - Существует несколько фреймворков внедрения зависимостей, проверенных на практике, — например, Spring, Dropwizard или Guice. Они предоставляют много полезных возможностей, но при этом полагаются на неявные допущения и вводят жесткое связывание в кодовую базу.
- Реактивное программирование обеспечивает функциональную обработку, управляемую данными, которая выполняется без блокирования, что позволяет ее распараллелить, разбив работу на несколько потоков.
 - Используя однопоточную обработку в качестве примера, мы изменили ее так, чтобы она проходила асинхронно и параллельно. Таким образом, мы можем ее распараллелить и обеспечить более высокую пропускную способность.
 - На основании последних тенденций мы переработали решение в реактивный поток данных, так как это соответствовало выполнению преобразований с N входными элементами.
 - Изучение потоковой модели всех решений поможет лучше анализировать достоинства и недостатки каждого из них.
- Функциональное программирование имеет множество преимуществ: более простую модель параллелизма, более компактный код, простоту тестиро-

вания и т. д., но бездумное применение функциональных средств в языках, оптимизированных для объектно-ориентированного программирования, может создать проблемы.

- Мы реализовали функциональную конструкцию на языке Java на базе рекурсии. Затем сравнили этот подход с хвостовой рекурсией в Scala.
- Неизменяемость — мощная концепция, но она имеет свою цену. После создания неизменяемый объект нельзя модифицировать. В качестве примера был реализован неизменяемый список.
- Так как приложения обычно взаимодействуют с несколькими компонентами, мы рассмотрели концепции отложенной и немедленной инициализации и присущие им компромиссы по времени инициализации, времени обработки запроса и обнаружению ошибок.

Томаш Лелек, Джон Скит

Software: Ошибки и компромиссы при разработке ПО

Перевел с английского Е. Матвеев

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Е. Строганова</i>
Литературный редактор	<i>М. Трусковская</i>
Художник	<i>В. Мостипан</i>
Корректоры	<i>С. Беляева, Г. Шкатова</i>
Верстка	<i>М. Жданова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Саппсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 04.2023. Наименование: книжная продукция.

Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,

58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 17.02.23. Формат 70х100/16. Бумага офсетная. Усл. п. л. 37,410. Тираж 700. Заказ