«Сегодня потоковые системы во многом изменились, но базовые характеристики остались прежними. В этой короткой книге я попытался извлечь ключевые элементы которые одинаково применимы и к системам распределения журналов, и к инструментам потоковой обработки данных.

— Бен Стопфорд

«Идеи, изложенные в этой книге, станут следующей ступенью нашего представления о совместном использовании и обмене данными, помогая нам изменить подход в построении микросервисных архитектур. Эти идеи могут показаться странными на первый взгляд, но их стоит придерживаться. Бен устроит вам очень интересное путешествие.

— Сэм Ньюман

ITSumma Press

itsumma.ru

books@itsumma.ru

fb.com/itsumma

vk.com/itsumma

При поддержке Tolma.ch

O'REILLY

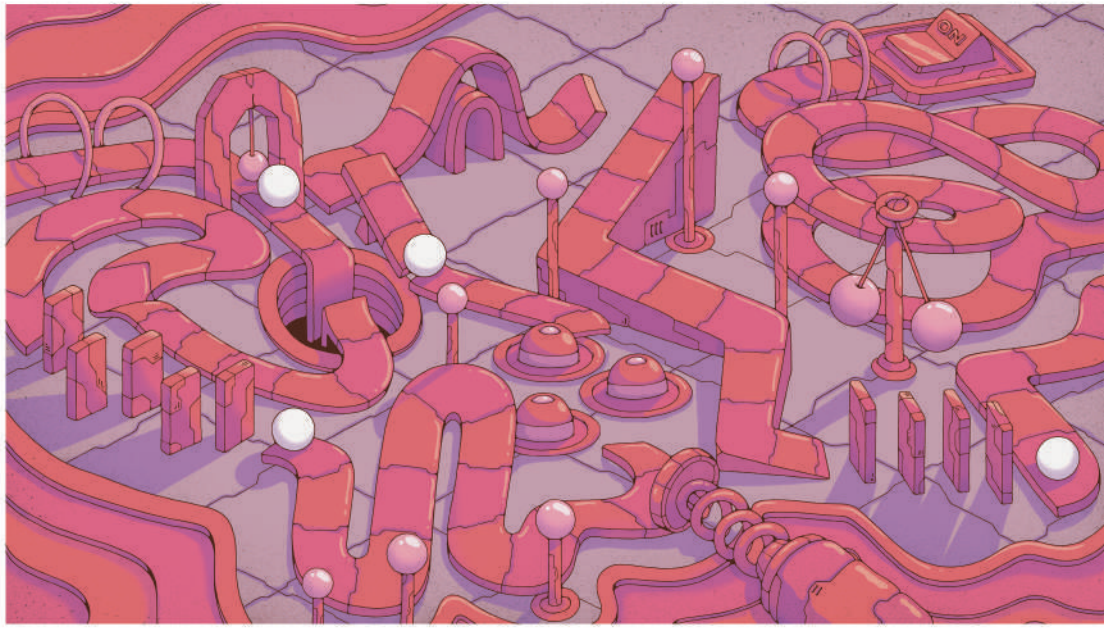
ISBN 978-5-6042412-1-9

9 785604 241219

Бен Стопфорд. Проектирование событийно-ориентированных систем

Проектирование событийно-ориентированных систем

Концепции и шаблоны проектирования сервисов потоковой обработки данных с использованием Apache Kafka



ITSumma Press

Бен Стопфорд

Ben Stopford

DESIGNING EVENT-DRIVEN SYSTEMS

Concepts and Patterns for Streaming Services with Apache Kafka

O'Reilly

Бен Стопфорд

ПРОЕКТИРОВАНИЕ СОБЫТИЙНО-ОРИЕНТИРОВАННЫХ СИСТЕМ

Концепции и шаблоны проектирования сервисов потоковой обработки данных с использованием Apache Kafka

Перевод с английского

Иркутск
ITSumma Press
2019

УДК 004.451
ББК 32.972.34
С81

Главный редактор	***Анастасия Овсянникова***
Научные редакторы	***Дмитрий Чумак, Иван Сидоров***
Перевод	***Владимир Жданов***
Корректоры	***Лариса Ковтун, Евгений Финкельштейн***
Верстка	***Неля Алькова***
Иллюстрация на обложке	***Иннокентий Астафьев***

Впервые на русском языке

Стопфорд, Бен

Проектирование событийно-ориентированных систем: Концепции и шаблоны проектирования сервисов потоковой обработки данных с использованием Apache Kafka / Бен Стопфорд ; Пер. с англ. — 2-е изд., испр. — Иркутск : ITSumma Press, 2019. – 175 с.

Современный мир плотно покрыт коммуникационными сетями разного уровня, и с лёгкой руки его можно назвать информационно связанным. Но, несмотря на это, всё ещё есть множество ситуаций, когда различным людям, командам и системам предпочтительнее взаимодействовать асинхронно, обмениваясь информацией лишь изредка, от случая к случаю, но при этом сохранять погружённость в общие контексты. Добиться этого в привычных императивных системах довольно сложно. Что делает подход, предлагаемый создателями Apache Kafka, особенно ценным. Данная книга поможет всем заинтересованным ознакомиться с общими принципами построения слабосвязанных событийно-ориентированных архитектур и раскроет основные преимущества и недостатки применяемых подходов.

ISBN 978-5-6042412-1-9

ISBN 978-1492038238
англ.

Оглавление

Вступительное слово от ITSumma Press

Мы с волнением и радостью представляем вам первую книгу нашего издательства. Почему мы, на сто процентов технологическая компания, решили заняться таким, казалось бы, консервативным направлением, как издание книг?

Многие считают, что книги уже не в моде. Однако у нас материальные источники информации вызывают большое уважение. Особенно когда это источник такой информации, которая не требует немедленного практического применения. Когда нет нужды с руководством на одном мониторе писать код на втором мониторе, а на третьем сразу проверять полученные результаты. Когда книга — это скорее ресурс для размышлений о подходах к технологиям, для комплексного анализа и выводов, к которым можно будет обращаться за советом еще долгое время. Чего не скажешь про прикладные учебники и руководства, быстро теряющие свою актуальность в современных реалиях. Эта книга будет интересна в течение многих лет после издания необычностью предлагаемых подходов к решению набивших оскомину проблем. Именно такого рода книги мы и хотели бы представить вниманию наших читателей.

Работа Бена Стопфорда стала нашим первым выбором не случайно. В ней автор позволяет взглянуть на устоявшиеся подходы к проектированию программных (веб-ориентированных и не только) систем с довольно непривычной для современного разработчика точки зрения. Как один из главных идеологов и создателей самой популярной на данный момент системы обмена сообщениями Kafka, Стопфорд вкладывает всю душу в то, чтобы завлечь читателя в свой мир бесконечного обмена событиями, асинхронных взаимодействий и конвейерной обработки данных.

Автор подвергает сомнению желание многих продолжать внедрять устоявшиеся монолитные схемы. Базы данных — это больше не центр вселенной, мир не крутится вокруг них. Десятки команд могут выпускать различные части единой системы совершенно независимо друг от друга, и при этом стабильность самой системы никак не пострадает.

Звучит несколько безумно? Или нет?

Ну и конечно же, будем рады обсудить с нашими читателями их впечатления, выводы, предложения. Возможно даже, кто-то захочет поделиться своими необычными «вечными» книгами, которые перевернули его взгляд на мир технологий, но до сих пор не были переведены на русский язык. Напишите нам о них.

До связи!

Предисловие от научного рецензента

В своей деятельности в компании ITSumma мы встречаемся и с огромным количеством технологий как таковых, и с самыми разнообразными архитектурными подходами. В том числе и с событийно-ориентированными системами, которые неразрывно связаны с микросервисной архитектурой. Последняя стала неотъемлемой частью индустрии.

Однако, кроме самого концепта разделения монолита на микросервисы, всё ещё остаются вопросы об их взаимодействии. Зачастую микросервисная архитектура выглядит как «технология ради технологии», в результате ухудшающая качество продукта. А переход на неё — как поиск «серебряной пули». В таком случае потоки данных между сервисами становятся во главу угла.

В нынешних условиях нескончаемого потока информации, микросервисы должны не просто общаться между собой по какому-то протоколу, но и обрабатывать гигантские потоки данных в реальном времени. Одно из направлений в проектировании таких систем — событийно-ориентированное взаимодействие между микросервисами. В своей книге Бен Стопфорд детально рассмотрел подходы по построению таких архитектур. В качестве основной системы автор приводит Apache Kafka — не просто как брокер сообщений, а как многофункциональный инструмент для микросервисного взаимодействия и обработки данных.

В книге изложены подходы, которые только начали применяться, несмотря на то, что теоретическая база уже достаточно хорошо проработана. На момент подготовки русскоязычного издания для большинства определений, упоминаемых автором, еще не было устоявшегося перевода с английского. Поэтому отдельное внимание мы уделили работе с терминологией и разъяснению некоторых понятий. Кроме того, используя наш повседневный опыт работы со схожими системами, мы постарались подойти к переводу книги не как к подготовке очередного типового технического издания, а вложить весь наш накопленный технологический опыт. Создать реально полезное, понятное, не пестрящее англицизмами и не искажающее смысл издание, листая которое хотелось бы углубиться в

чтение, а не ограничиться скептическим «лучше оригинал почитаю», как это обычно бывает с переводами технической литературы.

— Иван Сидоров
Исполнительный директор компании ITSumma

Вступление

Всё то долгое время, что мы говорим о сервисах, мы говорим о данных. На самом деле, прежде чем в нашем лексиконе появился термин «микросервисы», в старые-добрые времена сервис-ориентированных архитектур, мы говорили о данных: как получать к ним доступ, где их хранить, кто ими «владеет». И хотя данные являются главным фактором успеха бизнеса в долгосрочной перспективе, довольно часто они же становятся камнем преткновения при проектировании и развитии систем.

Моё собственное путешествие в мир микросервисов началось с работы, которую я выполнял, помогая различным организациям выпускать своё программное обеспечение (ПО) быстрее. В итоге я тратил огромное количество времени на такие задачи, как периодический анализ временных затрат, построение конвейеров сборки, автоматизацию тестирования и развёртывание инфраструктуры. Появление облаков стало огромным благом, так как автоматизация развёртывания инфраструктуры позволила нам быть ещё более продуктивными. Но некоторые фундаментальные проблемы всё же остались. Зачастую приложение было разработано так, что его было очень сложно выкладывать. И данные были ядром проблемы.

В то время самым частым приёмом, который мне доводилось видеть в сервис-ориентированных системах, было разделение одной базы данных (БД) между несколькими сервисами. Причины были просты: необходимые мне данные уже лежат в другой БД, к которой у меня есть доступ; поэтому мне оставалось лишь зайти в эту БД и взять то, что мне нужно. На начальном этапе это способствует быстрой разработке нового сервиса, но с течением времени такой подход становится источником больших проблем.

Как я подробно описывал в моей книге «Построение микросервисов»,[1] разделяемая БД создаёт в вашей архитектуре значительный узел зацепления. Становится сложно понимать, какие изменения можно вносить в структуру БД, разделяемой между множеством отдельных сервисов.

Дэвид Парнас[2] показал нам в 1971 году, что секрет разработки программного обеспечения, составные части которого могут меняться

независимо — это скрывать информацию этих частей друг от друга. Ведь, условно, разделение БД между различными сервисами ограничивает возможность независимого развития наших кодовых баз.

Вместе с изменением требований и ожиданий от ПО изменилась и организация ИТ-команд. Переход от разрозненных ИТ-департаментов к бизнес- или продукто-ориентированным командам помог улучшить клиенто-ориентированность этих команд. Этот переход часто происходил параллельно с желанием повысить автономность этих команд, позволить им развивать новые идеи и внедрять их, вместе с тем снижая необходимость координации с другими частями организации. Но сильно связанные архитектуры требуют значительной координации между системами и поддерживающими их командами и, следовательно, являются врагом любой организации, желающей повысить автономность.

В Amazon это поняли много лет назад. Они хотели повысить автономность команд разработчиков, чтобы позволить компании развиваться и разрабатывать ПО быстрее. И с этой целью были созданы маленькие, независимые команды, которые полностью контролировали весь цикл разработки ПО. Стив Йегге после своего ухода из Amazon в Google пытался ухватить идею того, что именно помогало группам разработчиков работать так хорошо в его печально известной (в определённых кругах) «Тираде о платформах (англ. Platforms Rant)».[3] В ней он выделил наказ основателя Amazon Джеффа Безоса относительно того, как команды должны взаимодействовать, и как они должны проектировать системы. И, в частности, эти пункты перекликаются с моими мыслями:

> 1) отныне все команды будут предоставлять свои данные и функции через сервисный интерфейс;
> 2) команды должны взаимодействовать друг с другом через эти интерфейсы;
> 3) никакой другой формы межпроцессного взаимодействия разрешено не будет: ни прямых ссылок, ни непосредственных чтений из баз данных другой команды, ни моделей разделяемой памяти, никаких обходных трюков. Единственным допустимым способом коммуникации остаются сетевые запросы через сервисный интерфейс.

Я же пришёл к осознанию того, что способ хранения и передачи данных имеет ключевое значение для разработки слабосвязанных архитектур.

Первостепенную важность имеют чётко определённые интерфейсы и сокрытие информации. Если нам надо хранить данные в базе, эта база должна быть частью сервиса, недоступной напрямую для других сервисов. Чётко определённый интерфейс должен описывать, когда и как данные можно получать и обрабатывать.

Большую часть последних нескольких лет я потратил на продвижение

этой идеи. Но, несмотря на то, что всё больше людей принимают эту идею, сложности остаются. В реальном мире сервисам нужно работать совместно и иногда необходимо делиться данными. Как сделать это эффективно? Как убедиться, что обмен данными выстроен с учетом требований к нагрузкам и сетевым задержкам вашего приложения? Что происходит, когда одному сервису требуется большое количество информации от другого?

Начните использовать потоки событий, а конкретнее — потоки вроде тех, что организовывают Kafka и похожие технологии. Мы уже используем брокеры для обмена событиями, но способность Kafka делать этот поток устойчивым позволяет нам рассмотреть новый способ хранения и обмена данных без потери возможности создания слабосвязанных автономных архитектур. В этой книге Бен говорит об идее «выворачивания базы данных наизнанку» — понятии, которое, я подозреваю, получит столько же скептических отзывов, сколько получил в своё время я, когда продвигал идею ухода от гигантских разделяемых БД. Но за последнюю пару лет, что я провёл, исследуя эти идеи с Беном, я не могу отделаться от мысли, что и он, и другие люди, работающие над этими понятиями и технологиями (конечно, с огромным количеством прототипов), действительно, что-то поняли.

Я надеюсь, что идеи, изложенные в этой книге, станут следующей ступенью нашего представления о совместном использовании и обмене данными, помогая нам изменить подход в построении микросервисных архитектур. Эти идеи могут показаться странными на первый взгляд, но к ним стоит присмотреться. Бен устроит вам очень интересное путешествие.

— Сэм Ньюман

Предисловие

В 2006 году я работал в компании ThoughtWorks в Великобритании. В то время наш офис был полон энергии, мы занимались кучей интересных вещей. Движение Agile было на пике, разработка через поведение (англ. behavior-driven development) процветала, люди экспериментировали с источниками событий (англ. event sourcing), а сервис-ориентированные архитектуры применялись в более мелких проектах, чтобы в итоге помочь разобраться с некоторыми из проблем, которые мы встречали в крупных проектах.

Одним из проектов, над которым я работал, руководил Дэйв Фарли, энергичный и весёлый парень, который смог привнести свою энергию практически во всё, что мы делали. Проект был относительно стандартным, средних размеров корпоративным приложением. Это был веб-портал, где клиенты могли запрашивать различные юридические услуги, касающиеся прав собственности на недвижимость. В ответ система запускала различные синхронные и асинхронные процессы, чтобы привести в действие множество сервисов.

В этом конкретном проекте был ряд интересных элементов, но единственной деталью, которая вводила меня в ступор, было то, как сервисы взаимодействовали между собой. Это была первая система, над которой я работал, построенная исключительно на событийном взаимодействии. Ранее я участвовал в разработке нескольких сервис-ориентированных систем, построенных на удалённых вызовах процедур (англ. remote procedure calls, далее — RPC) или системах обмена «запрос-ответ», и новый проект мне показался совершенно другим. Было что-то волшебное в том, что вы могли просто подключать новые сервисы прямо к потоку событий и с чувством глубочайшего удовлетворения наблюдать за тем, как система со свистом обрабатывает данные.

Несколько лет спустя я работал в крупной финансовой организации, которая хотела построить сервис данных в самом сердце компании, где приложения могли бы найти наборы данных необходимые для работы банка — сделки, оценки, справочные данные и тому подобное. Я считаю такого рода задачу достаточно привлекательной: она была технически сложной, и, хотя ряд банков и других крупных компаний использовали

такой подход и раньше, мне казалось, что технологии продвинулись к этапу, когда мы могли бы построить что-то действительно интересное и инновационное.

Но правильное понимание технологии было только началом решения проблемы. Система должна была взаимодействовать с каждым крупным отделом, что подразумевало огромное количество заинтересованных сторон с большим потоком требований, многочисленные расписания выпуска новых версий и разноплановые ожидания по продолжительности безотказной работы. Я помню, как мы обсуждали практические детали проекта, проговаривая его структуру во время двухнедельного установочного совещания. И хотя задача казалась довольно трудной не только технически, но и организационно, мы верили, что сможем ее выполнить.

В итоге мы собрали команду с кучей людей из ThoughtWorks, Google и нескольких других мест и создали систему с довольно интересными свойствами. Вычислительный кластер держал все запрашиваемые данные в памяти, распределённой среди более чем 35 машин в каждом дата-центре. Это позволяло выдержать запросы вычислительной сети. Записи шли напрямую через слой запроса в систему обмена сообщениями, которая создавала (что было несколько необычно для того времени) систему записи (англ. system of record). Как слой запроса, так и слой обмена сообщениями были спроектированы для дальнейшей сегментации, которая позволяла им линейно масштабироваться. Поэтому каждый запрос на вставку или изменение публиковался еще и как событие, без вариантов, ведь это было заложено в сердце архитектуры.

Интересно, что при превращении системы обмена сообщениями в систему записи поток данных можно преобразовать во множество полезных вещей: записывать его резервную копию на диск, перенаправлять его в другой дата-центр, формировать различные аналитические базы данных для построения отчетов, и, конечно, предоставлять доступ к потоку данных всем желающим, через какое-либо API (программный интерфейс приложения, англ. application programming interface).

Но в то время я еще не осознавал реальную важность использования системы обмена сообщениями в качестве системы записи. Я помню, как после моей презентации о проекте на конференции QCon один слайд «Обмен сообщениями как система записи», который я едва не перелистнул, зачитав лишь название, вызвал больше вопросов, чем весь материал о гармонично распределённом слое слияния, о котором я не замолкал всю презентацию. Так постепенно стало ясно, что большинству клиентов по-настоящему достаточно чего-то более простого, чем все эти штуки, типа предварительного кэширования, ориентированного на данные

и ускоряющего операции соединения (англ. join); интерфейса по типу SQL-через-Документ (англ. SQL-over-Document interface); неизменяемых моделей данных; и схем отложенного связывания. В то время как они начинали использовать сервис данных по прямому назначению, по прошествии времени нужда заставляла их делать копии, хранить их независимо и, в итоге, обрабатывать самостоятельно. Но, несмотря на это, они все-таки считали центральный набор данных целесообразным и часто создавали дополнительный поднабор, а затем возвращались для его расширения. Поэтому, поразмыслив, казалось, что система обмена сообщениями, оптимизированная для хранения наборов данных, будет более уместна, чем база данных, оптимизированная для их создания. Немного позже появился Confluent, и Kafka стала идеальным решением для этого типа проблем.

Интересно, что два этих проекта (приложение для купли-продажи и сервис общебанковских данных) имели больше общего, чем изначально могло показаться. Приложение для купли-продажи отлично поддерживало совместную работу и при этом его было легко встроить. В работе с банком, гораздо больший набор приложений и сервисов интегрировался между собой посредством событий, имея при этом возможность делать выборки по истории событий. И, несмотря на то, что предметные области проектов достаточно сильно отличались (небольшое приложение и ПО для целой компании), их общей особенностью был механизм использования событий.

Сегодня, потоковые системы во многом отличаются от вышеприведённых примеров, но базовые характеристики остались неизменными. Тем не менее, дьявол кроется в деталях, и за последние несколько лет мы видели, как клиенты использовали разнообразные подходы к решению подобного рода задач. В этой короткой книге я попытался извлечь ключевые элементы этих подходов, в которых одинаково активно применяются как системы распределённого журналирования (англ. distributed logs), так и инструменты потоковой обработки.

Как читать эту книгу

Книга состоит из пяти частей. Часть I базовая. В ней происходит знакомство с Kafka и потоковой обработкой данных и проводится полезный даже для опытных читателей обзор базовых понятий. В Части II вы узнаете, как строить событийно-ориентированные системы, как такие системы связаны с потоковой обработкой с состоянием (англ. stateful stream processing), и как применить такие шаблоны событийное сотрудничество, источник событий, и командно-запросная изоляция (англ. command query responsibility segregation, далее — CQRS). Часть

III носит больше теоретический характер, в ней мы возьмём основные идеи из Части II и приложим их к структуре целой организации. Здесь мы ставим под сомнение многие из распространённых ныне подходов и углубляемся в такие шаблоны, как Потоки событий как источник истины. Части IV и V практические. В Части V начинается первое погружение в программирование. Мы создали связанный проект на GitHub, который станет для вас стартовой площадкой, если вы захотите начать строить небольшие сервисы, используя Kafka Streams.

Вступление, данное в Главе 1, представляет поверхностный обзор основных понятий, затронутых в этой книге, так что это хорошее место для старта.

Благодарности

Многие люди внесли свой вклад в эту книгу, как прямо, так и косвенно, но особую благодарность хотелось бы выразить следующим людям: Джей Крепс, Сэм Ньюман, Эдвард Рибейро, Гвен Шапира, Стив Каунсел, Мартин Клепман, Ева Бизек, Ден Хэнли, Тим Бергланд, и, конечно же, моя всетерпящая жена Эмили.

ЧАСТЬ I

Расставляем декорации

Правдиво только то, что в журнале событий.

— Пэт Хелланд[4]

ГЛАВА 1

Введение

Хотя главной темой этой книги является построение событийно-ориентированных систем разных размеров, в основе лежит мысль о программном обеспечении, которое связывает многие команды. Это мир сервис-ориентированных архитектур: идея, возникшая в начале века, основанная на том, что компании перестраиваются вокруг использования разделяемых сервисов, которые делают простые и полезные вещи.

И идея эта стала весьма популярной. Amazon произвели фурор, запретив все внутрисистемные коммуникации с чем-либо, у чего не было сервисного интерфейса. Позже Netflix полностью переехал на микросервисы, и многие другие интернет-стартапы последовали их примеру. Корпорации делали схожие вещи, но часто с использованием систем обмена сообщениями со слегка иной динамикой. Мы многому научились за это время, достигли существенного прогресса, это нам далось нелегко.

Одним из уроков, довольно распространённым надо сказать, было то, что сервисный подход значительно увеличивает вероятность того, что вы будете разбужены в три часа ночи из-за того, что у вас упал один или несколько сервисов. И если задуматься, то не так уж это и удивительно. Если вы берёте набор в значительной степени независимых приложений и превращаете их в сеть тесно связанных, будет несложно представить, как один важный, но ненадёжный сервис может серьёзно повлиять на поведение всей системы, а то и привести к полному её падению. Как заметил Стив Йегги в своём знаменитом посте о Google и Facebook, «переход на сервисную архитектуру научил команды разработчиков не доверять друг другу почти так же сильно, как и любым разработчикам со стороны».[5]

Что на самом деле пошло на пользу Amazon, так это организационные изменения, которые привнёс полностью сервис-ориентированный подход. Команды, разрабатывающие сервисы, думали о своём ПО как о шестерёнке в очень большом механизме. Как выразился Иэн Робинсон,

«не будьте частью сети, будьте самой сетью». Это было серьёзным сдвигом по сравнению с тем, как люди строили приложения ранее, когда все внутрисистемные взаимодействия строились разработчиками по остаточному принципу. Но сервисная модель вывела это взаимодействие на передний план. Неожиданно вашими пользователями перестали быть только клиенты или бизнес-партнёры. Ими стали другие приложения, и им действительно важно, чтобы ваш сервис был надёжным. Так приложения стали платформами, а строить платформы непросто.

В LinkedIn почувствовали эту боль, когда перешли со своего изначального, монолитного Java-приложения на несколько сотен отдельных сервисов (от 800 до 1100). Сложные зависимости не добавляли стабильности, проблемы версионирования приводили к болезненным и топорным выкладкам, и поначалу было совсем не очевидно, что новая архитектура на самом деле была большим шагом вперёд.

Основным отличием на пути эволюции LinkedIn было использование системы обмена сообщениями собственной разработки — Kafka. Kafka добавила в общую архитектуру асинхронную модель издатель-подписчик, позволившую передавать внутри организации триллионы сообщений в день. Это было важно для компании в стадии активного роста, т. к. позволяло подключать новые приложения, не затрагивая хрупкую сеть синхронных взаимодействий, на которых держался пользовательский интерфейс.
Но идея перестройки систем вокруг событий не нова — событийно-ориентированные архитектуры окружают нас десятилетиями, и технологии, типа систем обмена сообщений корпоративного уровня, это большой бизнес, в частности, среди (что неудивительно) корпораций. Многие корпорации существуют уже довольно давно, и их системы росли органически, проходя большое количество итераций или через внедрение сторонних решений. Системы обмена сообщениями естественным образом вписываются в эти сложные и разделённые миры по тем же причинам, что обнаружили в LinkedIn: события изолируют, и это значит, что разные части компании могут работать независимо друг от друга. Это также означает, что будет гораздо проще подключать к потокам событий реального времени новые системы.

В качестве отличного примера применения подобных систем можно привести ограничение, наложенное на финансовую отрасль в январе 2018.[6] Оно заключается в том, что о любой трейдерской активности нужно уведомлять регулятора в течение одной минуты после заключения сделки. Минута может показаться довольно продолжительным отрезком времени в вычислительном плане, но достаточно всего одной пакетно-ориентированной системы в критической точке в одном подразделении компании, чтобы не уложиться в отведённое время. Поэтому банки,

вложившие усилия во внедрение систем обработки торговых событий реального времени и подключившие к ним все свои подразделения, легко вписались в наложенные ограничения. Для большинства же, у кого подобных систем не было, соответствие стандартам стоило значительных усилий, обычно выливавшихся в половинчатые и временные решения.

Корпорации всегда были сложными и разделёнными: много отдельных, асинхронных островков, работающих независимо друг от друга большую часть времени, часто каждый со своими отдельными пользователями. Интернет-компании же другие, они обычно зарождаются простыми, ограничиваясь витриной веб-приложения, в котором пользователи нажимают на кнопки и ожидают какого-либо результата. Многие начинают как монолиты и остаются такими на какое-то время (пожалуй, на более длительное, чем следовало бы). Но по мере роста интернет-компаний, их деятельность становится более сложной, и они тоже начинают смещаться в сторону асинхронности. В компанию включаются новые команды и отделы, и им нужно действовать независимо, свободно от синхронных уз, связывающих пользовательский интерфейс. Так, повсеместная тяга к онлайн-утилитам для платежей или для обновления корзины постепенно сменилась растущей потребностью в наборах данных, которые можно использовать и развивать без привязки к какому-то конкретному приложению.

Но системы обмена сообщениями — не панацея. У сервисных шин предприятия (англ. enterprise service bus, далее — ESB), к примеру, множество противников,[7] а традиционные системы обмена сообщениями имеют ряд собственных проблем. Они часто используются для перемещения данных внутри организации, но отсутствие в них понятия о хронологии ограничивает их значимость. Так, даже если недавние события важнее, чем более старые, бизнесу обычно всё ещё требуются исторические данные: будь то пользователь, которому нужна история действия с аккаунтом, или сервис, которому требуется список клиентов, или аналитики, которым нужно построить отчёт для начальства.

С другой стороны, сервисы данных с HTTP-интерфейсом делают выборки простыми. Любой может подключиться и выполнить запрос. Но они не облегчают сам процесс передачи данных. Чтобы извлечь данные, вы отправляете запрос и потом периодически опрашиваете сервис на предмет изменений. Это выглядит как временное решение, и наверняка администраторы, ответственные за работу сервиса, который вы постоянно дёргаете, не скажут вам спасибо.

Но воспроизводимые журналы, как у Kafka, могут взять на себя обязанности источника событий — прослойки между системой обмена сообщениями и базой данных. (Если вы пока ничего не знаете о Kafka,

не беспокойтесь — мы изучим её детально в Главе 4) Воспроизводимые журналы отделяют сервисы друг от друга так же, как это делает система обмена сообщениями, но кроме этого, они также предоставляют единое хранилище, отказоустойчивое и масштабируемое — общий источник истины, на который может сослаться любое приложение.

А общий источник истины оказывается удивительно полезной вещью. Микросервисы, к примеру, не разделяют свои базы данных друг с другом (так называемый антишаблон интегрирующей БД, англ. IntegrationDatabase antipattern).[8] Для этого есть веская причина: базы данных обладают богатым API, которое удобно само по себе, но когда его использует сразу несколько сервисов, то становится сложнее понять повлияет ли одно приложение на другое и каким образом (будь то зацепление по данным, проблемы с синхронизацией или нагрузка). Но бизнес факты, которыми сервисам всё же приходится обмениваться, являются наиболее важными данными из всех. Они являются истиной, на которой строится остальная часть бизнеса. Пэт Хэлланд указывал на это различие ещё в 2006 году, назвав его «внешние данные».[9]

Но воспроизводимый журнал предоставляет куда более подходящее место для подобных данных, потому что (как бы парадоксально это не звучало) вы не можете выполнить к нему запрос! Это чистое хранение данных и их последующая передача в новое место. Идея чистой передачи данных очень важна, ведь чем больше внешних данных, данных, которые сервисы делят между собой, тем более плотно они друг с другом связаны, и чем больше сервисов в экосистеме, тем плотнее связываются данные. Решением является вынос данных туда, где связи будут слабее. Другими словами, нужно перенести данные в приложение, где вы сможете ими манипулировать как вашей душе угодно. Так что перемещение данных даёт приложениям уровень работоспособности и контроля, недостижимый при использовании прямых зависимостей времени исполнения. Идея сохранения контроля оказалась важной, по этой же причине шаблон разделяемых баз данных на практике работает не так хорошо.

Таким образом, у подхода, ориентированного на воспроизводимые журналы, есть два основных преимущества. Во-первых, он позволяет реагировать на события, происходящие в данный момент, с помощью набора инструментов, специально предназначенных для манипулирования ими. Во-вторых, он обеспечивает центральное хранилище, которое может отправлять целый набор данных туда, где он может потребоваться. Это очень полезно, когда центры обработки данных компаний расположены по всему миру, когда вам необходимо быстро запустить или прототипировать новый проект, сделать некоторые специальные исследования данных, или построить сложную экосистему сервисов, которые будут развиваться свободно и независимо.

Таким образом, существуют явные преимущества событийно-ориентированного подхода (как, разумеется, существуют преимущества и у моделей REST/RPC). Но это, по сути, только половина истории. Поток данных — это не просто альтернатива RPC, которая работает лучше в сильно связанных случаях. Это фундаментальное изменение в мышлении, которое включает переосмысление бизнеса как развивающихся потоков данных, а сервисов — как функций, которые трансформируют эти потоки в нечто новое.

Это может показаться странным. Многие из нас обучены программированию как процессу задавания вопросов или выдачи команд, и ожидания ответа. Так работает процедурное или объектно-ориентированное программирование, но главной причиной сложившегося восприятия, пожалуй, стоит считать базы данных. Более полувека базы данных играли центральную роль в архитектуре систем, формируя стиль написания (и осмысления) программ больше, чем какой-либо другой инструмент. В некотором смысле, это сыграло злую шутку.

По мере продвижения от главы к главе, эта книга построит картину несколько иного подхода к обработке данных, где база данных разобрана на составляющие и вывернута наизнанку. Эти идеи могут показаться странными и чужеродными, но, как и многие другие концепции в программном обеспечении, они являются лишь эволюцией старых идей, развившихся из разных технологических субкультур. В течение некоторого времени программисты используют событийно-ориентированные архитектуры, источник событий, и CQRS в качестве мер для решения проблем, связанных с масштабированием систем, работающих вокруг баз данных. В сфере больших данных сталкиваются с похожей проблемой: пакетная обработка на наборах данных во многие терабайты крайне непрактична.[10] И это приводит к мысли о потоковой обработке. Функциональный мир стоит в стороне, возможно, осознанно, и периодически озирается на императивные взгляды масс.[11]

Но этот разрозненный процесс выворачивания баз данных наизнанку, их разбор, и CQRS имеют нечто общее. Все они выражают стремление к тому, что нужно разделить концепции, встроенные во все базы данных, разобрать их и использовать их части отдельно, более эффективно.

Для этого существует множество причин, но, пожалуй, главная из них состоит в том, что этот подход позволяет создавать большие и разнообразные по функциональности системы. И хотя подход, ставящий в центр мироздания базу данных, хорошо работает для отдельных приложений, мы не живём в мире отдельных приложений. Нас окружают взаимосвязанные системы, отдельные компоненты которых, несмотря на свою индивидуальную полезность, являются лишь частью огромной

мозаики. Нам нужен механизм обмена данными, который дополнит этот сложный и взаимосвязанный мир. События ведут нас к этому. Они без конца передают данные в наши приложения. Приложения реагируют, смешивают потоки, создают представления, изменяют состояния и двигаются дальше. В потоковой модели нет общей базы данных. Поток событий это и есть база данных, и приложения лишь формируют поток в нечто новое.

Справедливости ради, потоковые системы по-прежнему обладают признаками баз данных, такими как таблицы (для поиска) и транзакции (для атомарности), но сам подход отличается кардинально, приближаясь к функциональному языку или языку потока данных (и между этими двумя мирами постоянно происходит «перекрёстное опыление»).

Поэтому, когда речь заходит о данных, мы должны быть твёрдо уверены в общих данных нашей системы. В конце концов, это основа нашего бизнеса. Факты могут развиваться со временем, применяться в разных аспектах, или вписываться в разные контексты, но они всегда должны отсылать к одной нити бесповоротной истины, из которой происходит всё остальное. Это эдакая центральная нервная система, которая лежит в основе любого современного цифрового бизнеса.

Данная книга рассматривает применение Apache Kafka в свете этой проблемы. В Части I мы познакомимся с потоками данных и посмотрим, как работает Kafka. В Части II мы сфокусируемся на шаблонах и техниках, необходимых для создания событийно-ориентированных программ: рассмотрим источники событий, событийное сотрудничество, CQRS и прочее. В Части III эти идеи разовьются дальше. Мы применим их в контексте многокомандных систем, включая микросервисы и сервис-ориентированная архитектура (англ. service-oriented architecture, SOA), сконцентрируемся на потоках событий, как источнике истины.[12] И проверим вышеупомянутую идею о том, что и системы, и компании могут быть переосмыслены вместе с выворачиванием БД наизнанку.[13] Заключительная часть будет более практической. Мы построим небольшую потоковую систему с помощью Kafka Streams (и KSQL).

ГЛАВА 2

Истоки потоковой обработки

Эта книга о том, как выстроить бизнес-системы с помощь инструментов потоковой обработки. Поэтому стоит уделить немного внимания происхождению данного подхода. Развитие этой группы инструментов в мире, которым движет аналитика в реальном времени, сильно повлияло на то, как мы теперь строим событийно-ориентированные системы.

Рис. 2.1 показывает систему потоковой обработки, которая получает данные с нескольких сотен тысяч мобильных устройств. Каждое устройство отправляет небольшое сообщение в формате JSON для обозначения приложений на мобильном телефоне, которые открываются, закрываются или падают. Так можно искать нестабильные места, где соотношение падений к нормальному использованию сравнительно высоко.

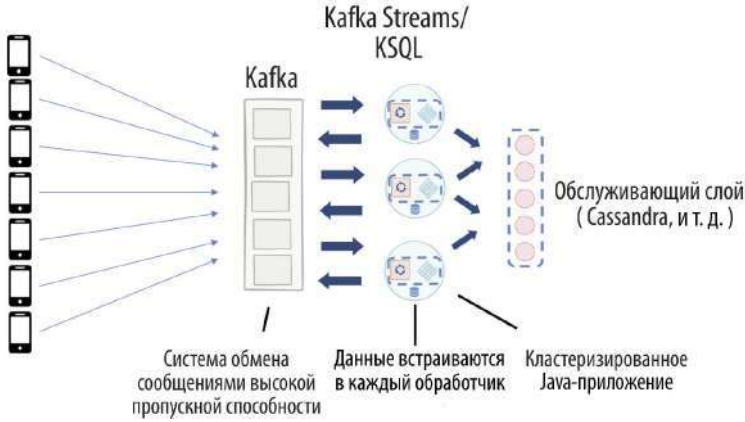


Рис. 2.1. Типичное потоковое приложение, которое переносит данные из мобильных устройств в Kafka, обрабатывает их в потоковом слое и отправляет в обслуживающий слой, где они могут быть запрошены

Мобильные устройства отправляют свои данные в Kafka, которая буферизирует их до того момента, когда они потребуются другим приложениям. Для такого типа нагрузки необходим относительно большой кластер. Kafka потребляет данные примерно со скоростью сети, но затраты на репликацию обычно делят скорость на три: так, кластер из трёх узлов на 10 GbE на практике будет переваривать примерно 1 Гбит/с. Справа от Kafka на рис. 2.1 находится слой потока обработки. Это кластерное приложение, где запросы могут быть либо определены изначально через Java DSL, либо отправлены динамически через KSQL[14] — язык SQL-подобной обработки процессов. В отличие от традиционных баз данных, эти запросы обрабатываются непрерывно, и с каждым входным сигналом, попадающим в обрабатывающий слой, запрос вычисляется повторно, и результат посылается, если значение изменилось.

Как только сообщение пройдёт все потоковые вычисления, результат попадёт в обслуживающий слой, который может быть запрошен. На рис. 2.1 показана Cassandra, но также распространена запись в HDFS (англ. Hadoop Distributed File System), в другой источник данных, или Kafka Streams с использованием интерактивных запросов.[15]

Чтобы лучше понять поток данных, давайте посмотрим на типичный запрос. Рис. 2.2 показывает поток, который вычисляет общее количество падений приложения в день. С каждым новым сообщением, говорящем о падении приложения, счётчик общих падений увеличивается на единицу. Заметим, что эти вычисления требуют определенного состояния: количество прошедших дней (например, в пределах длительности окна) должно быть сохранено, чтобы при падении или перезапуске потокового процессора счёт мог продолжаться с того же самого места. Kafka Streams и KSQL управляют этим состоянием внутри, и это состояние сохраняется в Kafka через тему Изменений (англ. changelog topic). Мы обсудим это более подробно в главе «Оконные функции, соединения, таблицы и хранилище состояний» на стр. 150 в Главе 14.

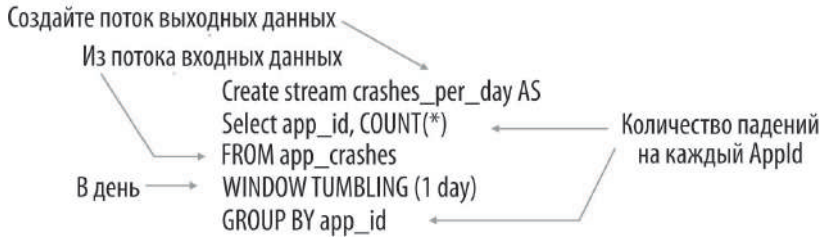


Рис. 2.2. Простой KSQL запрос, проверяющий количество падений в день

Множественные запросы этого типа могут быть соединены в конвейер (англ. pipeline). На рис. 2.3 мы разбиваем предыдущую проблему в три шага, соединённых в два этапа. Запросы (а) и (б) последовательно высчитывают суточные данные по открытым и упавшим приложениям соответственно. Два получившихся исходящих потока объединяются на финальном этапе (в), который вычисляет стабильность приложений через отношение падений к использованию и сравнение с фиксированной привязкой.

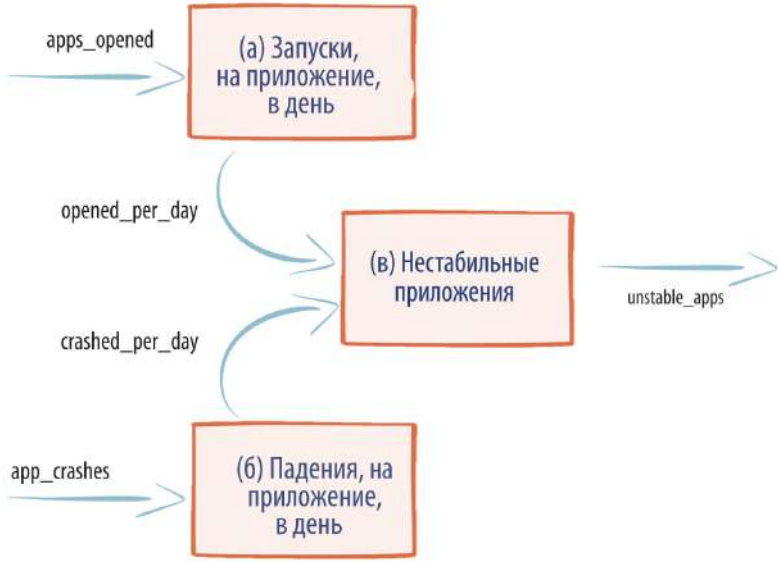


Рис. 2.3. Два первоначальных потока запросов собираются в третий поток, формируя конвейер

Есть ещё несколько замечаний о потоковом подходе, о которых стоит упомянуть:

Потоковый слой устойчив к отказам.

Он работает как кластер на всех доступных узлах. Если один узел выпадает, то другой продолжит его работу с того же самого места. И наоборот, можно масштабировать кластер путем добавления новых обрабатывающих узлов. Задачи и все необходимые состояния будут автоматически перенаправлены на использование этих ресурсов.

Каждый узел потокового обработчика может содержать собственное состояние

Это необходимо для буферизации или хранения целых таблиц, например, чтобы производить улучшение данных (потоки и таблицы мы разберём подробнее в разделе «Оконные функции, соединения, таблицы и хранилище состояний» на стр. 150 в

Главе 14). Идея локального хранилища важна, поскольку она даёт обработчику потока возможность производить быстрые запросы по входящему сообщению, не опрашивая соседние узлы — это необходимая функция для высокоскоростных операций и систем. Но эта способность сохранять состояния в локальных хранилищах также оказывается полезна для целого ряда бизнес-задач, что мы рассмотрим в этой книге чуть позже.

Каждый потоковый обработчик может записывать и сохранять локальное состояние

Опрашивать соседние узлы для обработки каждого входящего сообщения может быть не слишком хорошей идеей при работе с потоковыми системами большой пропускной способности. Именно поэтому потоковые обработчики записывают данные локально (для быстрой записи и чтения) и резервируют эти записи в Kafka. К примеру, упомянутый выше счётчик требует общего отслеживания так, чтобы при сбое или перезапуске, вычисления продолжились с предыдущей позиции, и счётчик оставался корректным. Эта возможность локального хранения данных концептуально очень похожа на взаимодействие с базой данных в традиционных приложениях. Но в отличие от традиционного двухуровневого приложения, где взаимодействие с базой данных подразумевает сетевой запрос, в обработке потоков всё состояние локально (воспринимайте его в виде кэша), поэтому доступ к нему происходит быстро, сетевые запросы не требуются. Поскольку оно также сбрасывается обратно в Kafka, оно наследует и гарантию долговечности Kafka. Мы обсудим это более подробно в разделе «Масштабирование параллельных операций в потоковых системах» на стр. 156 в Главе 15.

ГЛАВА 3

Всё ли вы знаете о Kafka?

Есть старая добрая притча о слоне и слепцах. Ни один из слепцов никогда раньше не встречался со слоном. Один слепец коснулся ноги слона и воскликнул: «Какой высокий! Он похож на дерево». Другой слепец дотронулся до хвоста и заявил: «Он похож на веревку». Третий потрогал хобот и сказал: «Он похож на змею». Так каждый слепой человек воспринимает слона по-своему и по-своему решает для себя, что такое слон. Конечно, слон похож на всё то, что упомянули слепцы, но в действительности это просто слон! Точно так же, когда люди узнают о Kafka, они часто видят её лишь с одной стороны. И хотя это позволяет детально изучить каждую отдельную сторону Kafka, полного понимания о всей платформе это не даёт. В этой главе мы рассмотрим Kafka с нескольких сторон.

Kafka это REST, только асинхронный?

Kafka предлагает асинхронный протокол для соединения программ, но он, безусловно, немного отличается от, скажем, протокола TCP, HTTP или протокола RPC. Разница в наличии брокера. Брокер представляет собой отдельный элемент инфраструктуры, который транслирует сообщения в любые программы, заинтересованные в них, а также хранит их настолько долго, насколько это необходимо. Поэтому он идеально подходит для потоковой доставки сообщений и доставки по принципу «послал и забыл».

Другие случаи использования выходят далеко за рамки первоначальных задач. Например, парадигма «запрос-ответ». Скажем, у вас есть сервис для запроса информации о покупателях. Он содержит вызов метода `getCustomer()`, получает `CustomerID` и в ответ выдает документ с описанием покупателя. С Kafka вы можете построить такой тип взаимодействия «запрос-ответ» с помощью двух тем, одна из которых доставляет запрос,

а другая — доставляет ответ. Некоторые строят подобные системы, но брокер в таких случаях не особо полезен. Ведь нет нужды в рассылке информации. И нет необходимости в её хранении. Возникает вопрос: может лучше использовать протокол без сохранения состояния, такой как HTTP?

Итак, Kafka — это механизм для обмена информацией, но его первоначальная идея состоит в событийной коммуникации, где события представляют собой бизнес факты, имеющие значение для нескольких служб и заслуживающие сохранения.

Похожа ли Kafka на сервисную шину?

Если рассматривать Kafka как систему обмена сообщениями, с её платформой Connect, способной запрашивать и передавать данные в контексте широкого круга интерфейсов и хранилищ данных, а также её потоковыми API, которые могут оперировать данными на лету, то это действительно немного похоже на ESB. Разница в том, что ESB фокусируется на интеграции унаследованных и новых систем, используя недолговечный и сравнительно низкопроизводительный слой передачи сообщений, который заставляет использовать протоколы типа «запрос-ответ» (см. предыдущий раздел).

Kafka, тем не менее, представляет собой платформу доставки потоков данных и, как положено таковой, делает акцент на высокоскоростных событиях и потоковой обработке. Кластер Kafka — это распределённая горизонтально-масштабируемая система, которая обеспечивает наращивание пропускной способности и поддерживает хранение данных. Это сильно отличается от традиционных систем обмена сообщениями, ограничивающихся одной машиной и если предполагают масштабирование, то не сквозное. Такие инструменты, как Kafka Streams и KSQL, позволяют писать простые программы, которые манипулируют событиями по мере того, как те появляются и развиваются. Данные инструменты позволяют пользоваться такими же возможностями, которые предоставляют базы данных, на уровне приложения, через API, без каких-либо ограничений общего брокера. Это крайне важно.

В некоторых кругах ESB не жалуют.[16] Критикуют в основном то, как технология развивалась в течение последних 15 лет, особенно тот факт, что центральные команды контролируют ESB и навязывают схемы, потоки сообщений, валидацию и даже преобразования. На деле подобный централизованный подход может ограничивать организацию, не позволяя отдельным приложениям и сервисам развиваться в своём собственном темпе.

В ThoughtWorks недавно обратили на это внимание,[17] призвав пользователей не воспроизводить проблемы ESB в работе с Kafka. В то же время, компания предлагает пользователям рассматривать потоки событий как источник истины,[18] о котором мы поговорим в Главе 9. Оба совета разумны.

И хотя Kafka, возможно, немного похожа на ESB, она, как мы увидим в этой книге, во многом отличается. Она обеспечивает гораздо более высокую пропускную способность, высокую доступность и долгосрочное хранение, поэтому сотни компаний направляют свои основные потоки данных в единый кластер Kafka. Кроме того, потоковая обработка данных побуждает сервисы сохранять контроль, особенно над своими данными, а не передавать управление отдельной центральной группе или платформе. И хотя иметь один отдельный кластер на Kafka в центре всей организации уже не считается чем-то необычным, популярность метода вызвана именно его простотой — ничего лишнего, кроме высокоскоростной и высокодоступной передачи и хранения данных. Это подчеркивает и главная мантра событийно-ориентированных сервисов. — Централизуй постоянный поток данных. Децентрализуй свободу действовать, адаптировать и менять.

Похожа ли Kafka на базу данных?

Некоторые люди любят сравнивать Kafka с базой данных. Kafka, несомненно, имеет схожие характеристики. Она предоставляет хранение, рабочие темы с сотнями терабайт данных в ней не редкость. Kafka имеет SQL интерфейс, позволяющий пользователям формировать запросы и выполнять их по данным в журнале. Их можно собрать в представление, которое пользователь в состоянии запросить напрямую. К тому же Kafka поддерживает транзакции. Всё это звучит вполне «по-базаданному»!

Несмотря на то, что многие элементы традиционной базы данных присутствуют в Kafka, она, если уж сравнивать, есть ничто иное, как база данных наизнанку (см. «База данных наизнанку» на стр. 95 в Главе 9) — инструмент для хранения и обработки данных в реальном времени, и создания представлений. Несмотря на то, что вы вполне вправе загрузить набор данных в Kafka, выполнить по нему запрос KSQL и получить ответ — как и в традиционной базе данных — KSQL и Kafka Streams оптимизированы для непрерывного вычисления, а не пакетной обработки.

Хотя в этой аналогии с базой данных есть доля правды, она немного не по делу. Kafka предназначена для передачи данных и одновременной обработки этих данных на лету. Во главе угла обработка данных в реальном времени, и только потом — долгосрочное хранение.

Что такое Kafka на самом деле? Платформа потоковой обработки данных

Как показывает рис. 3.1, Kafka это платформа потоковой обработки данных.[19] В её центре находится кластер брокеров Kafka (подробно рассматриваемый в Главе 4). Взаимодействовать с кластером можно через широкий спектр клиентских API на Go, Scala, Python и других языках.

Есть еще два API для потоковой обработки: Kafka Streams и KSQL (которые мы обсудим в Главе 14). Это движки базы данных для обработки данных на лету, позволяющие пользователям фильтровать потоки, соединять их вместе, группировать, хранить показатели состояния и выполнять произвольные функции с развивающимися потоками информации. Эти API могут иметь элементы состояния. Это означает, что они могут содержать таблицы данных совсем как обычная база данных (см. «Делаем сервисы контекстно-зависимыми» на стр. 53 в Главе 6).

Третий API — это Connect.[20] Он имеет целую экосистему интерфейсов для взаимодействия с различными типами баз данных или другими конечными точками как для извлечения данных из Kafka, так и для передачи данных в Kafka. Наконец, есть набор утилит, таких как Replicator и Mirror Maker, которые связывают воедино разрозненные кластеры, и реестр схем (англ.

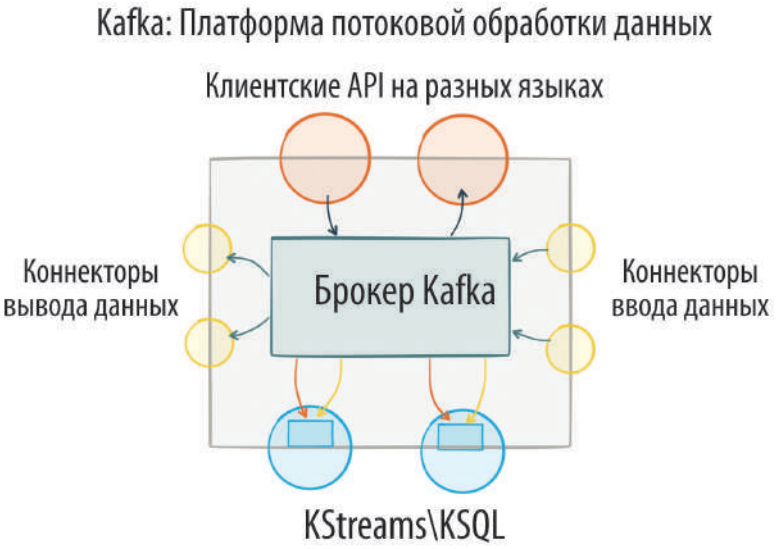


Рис. 3.1. Основные компоненты платформы потоковой обработки данных

Schema Registry), который утверждает схемы и управляет ими. Данные утилиты применяются к сообщениям, прошедшим через Kafka и ряд других инструментов в платформе Confluent.

Платформа потоковой обработки данных совмещает эти инструменты ради того, чтобы преобразовать хранимые данные (англ. data at rest) в данные, которые циркулируют в организации. Очень часто её сравнивают с центральной нервной системой человека. Способность брокера масштабировать, сохранять данные и работать бесперебойно делает его уникальным инструментом для соединения множества разрозненных приложений и сервисов в рамках отдела или организации. Интерфейс Connect позволяет легко уйти от устаревших систем, открывая скрытые наборы данных и превращая их в потоки событий. Потоковая обработка данных позволяет приложениям и сервисам встраивать логику напрямую в эти потоки событий.

ГЛАВА 4

Не только обмен сообщениями: обзор брокера Kafka

Кластер Kafka — это по сути своей совокупность файлов с сообщениями, распределённых по множеству различных машин. Большая часть кода Kafka направлена на связывание этих различных отдельных журналов друг с другом, надежную маршрутизацию сообщений от издателей к потребителям, создание реплик для увеличения отказоустойчивости и корректную работу с отказами. Таким образом, кластер Kafka — это система обмена сообщениями, по крайней мере, в некотором смысле, но он сильно отличается от тех брокеров сообщений, которые ему предшествовали. Как и любая технология, он имеет и плюсы, и минусы, формирующие впоследствии структуру системы, которую мы пишем. В этой главе мы рассмотрим брокер Kafka (т. е. серверный компонент) с точки зрения построения бизнес-систем. Мы вкратце рассмотрим, как он работает и углубимся в редкие сценарии использования, которые он поддерживает, например, хранение данных, динамическое переключение и защита полос пропускания (англ. bandwidth protection).

Первоначально задуманная для распределения наборов данных, созданных крупными социальными сетями, Kafka преимущественно формировалась необходимостью работать в больших масштабах в условиях высокого риска отказов. Соответственно, её архитектура берёт больше от систем хранения данных, например, HDFS, HBase и Cassandra, чем от традиционных систем обмена сообщениями, которые поддерживают JMS (англ. Java Message Service)[21] или AMQP (англ. Advanced Message Queuing Protocol).[22]

Как и многие положительные примеры в информатике, подобная масштабируемость обусловлена прежде всего простотой. Базовая абстракция здесь — это секционированный журнал (англ. partitioned log)[23] — т. е. набор дополняющихся (англ. append-only) файлов, распределенный по нескольким машинам. Такой подход поощряет схемы последовательного доступа, естественно сочетающиеся с сутью базового

аппаратного обеспечения.

Кластер Kafka — это распределённая система, которая рассредотачивает данные среди многих машин как для отказоустойчивости, так и для горизонтального масштабирования. Система спроектирована для работы в широком спектре случаев: от высокопроизводительной потоковой обработки данных, где важны только последние сообщения, до критически важных сценариев использования, в которых сообщения и их относительный порядок поступления должны быть сохранены с теми же гарантиями, какие вы могли бы ожидать от СУБД или системы хранения данных. Расплата за такую масштабируемость состоит в более простом протоколе, без таких дополнительных возможностей, как, например, селекторы сообщений в JMS или AMQP.

Но такая перемена курса имеет очень важные последствия. Параметры пропускной способности Kafka делают передачу данных от процесса к процессу быстрее и практичнее по сравнению с предыдущими технологиями. Способность Kafka хранить наборы данных устраняет проблемы глубины очереди,[24] которые отравляют жизнь традиционным системам обмена сообщениями. Наконец, ее богатый выбор API-интерфейсов, особенно Kafka Streams и KSQL, предлагает уникальный механизм для встраивания обработки данных непосредственно внутрь клиентских программ. Данные преимущества стали главной причиной того, что огромное количество разнонаправленных компаний используют Kafka в качестве несущего каркаса для обмена и хранения данных своих сервисов.

Журнал: эффективная структура для сохранения и распространения сообщений

В основе системы обмена сообщениями Kafka лежит секционированный, воспроизводимый журнал.[25] Идея проста: у вас есть набор сообщений, последовательно дописываемых в файл. Когда сервис хочет прочитать сообщения из Kafka, он ищет позицию последнего прочитанного им сообщения, и с этого места читает сообщения по порядку, одновременно фиксируя свою новую позицию в журнале (см. рис. 4.1).

Подход с использованием журнала имеет интересный побочный эффект. И чтение, и запись являются последовательными операциями. Это позволяет им органично вписываться в принципы работы находящегося уровнем ниже аппаратного обеспечения, использовать предварительные выборки, различные виды кэширования и естественным образом группировать операции. Это в свою очередь делает их эффективными. В самом деле, когда вы читаете сообщения из Kafka, сервер даже не импортирует их в

JVM (виртуальная машина Java). Данные копируются напрямую из дискового буфера в сетевой (zero copy). Это стало возможным благодаря простоте как протокола, так и базовой структуры данных.

Рис. 4.1. Журнал — это книга записей, доступная только для добавления данных

Таким образом, пакетированные, последовательные операции способствуют общей производительности. Они также адаптируют систему для более длительного хранения сообщений. Большинство традиционных брокеров сообщений строятся с помощью индексных структур — хэш-таблиц и B-деревьев, которые используются для управления уведомлениями, фильтрации заголовков сообщений и удаления сообщений после прочтения. Но минус в том, что эти индексы требуют должного обслуживания, отнюдь не бесплатного. Для лучшей производительности и быстрого отклика, индексы должны постоянно храниться в памяти, чтобы избежать пересчета индексов.[26] Сложность операций над журналом — O(1), будь то чтение с секции или запись в секцию, поэтому становится не так важно хранятся ли данные на диске или они закэшированы в памяти.

Секции и секционирование

Секции являются фундаментальным понятием для большинства систем распределённых данных. Секция — это просто ячейка, в которой размещаются данные, подобно ячейкам для группировки данных в хэш-таблице. В терминологии Kafka каждый журнал является репликой секции, размещённой на другой машине. (Так одна секция может иметь три реплики для обеспечения высокой доступности. Каждая реплика представляет собой отдельный журнал с теми же данными внутри.) Издатель (англ. Kafka producer) с помощью встроенного инструмента секционирования определяет какие данные отправляются в каждую из секций. Для распределения данных по доступным секциям будет использоваться циклический алгоритм, или, если вместе с сообщением был передан ключ, Kafka вычислит хэш-значение ключа и отправит данные в конкретную секцию. Последний вариант гарантирует, что все сообщения с одинаковым ключом попадут в одну секцию и, соответственно, будут жёстко упорядочены.

В журнально-ориентированном подходе (англ. log-structured approach)[27] есть несколько преимуществ. Если у сервиса произошёл перебой в работе, и он не читал сообщения в течение длительного времени, то объем накопленных непрочитанных сообщений не приведёт к значительному замедлению инфраструктуры (как обычно случается с традиционными брокерами, у которых снижается производительность с ростом очереди). Будучи журнально-ориентированной, Kafka хорошо подходит для исполнения роли источника событий (англ. event sourcing)[28] для тех, кто любит применять подобный шаблон в своих сервисах. Этот вопрос подробно рассматривается в Главе 7.

Линейная масштабируемость

Как мы уже обсуждали, журналы предоставляют структуру данных, органично вписывающуюся в принципы работы аппаратного обеспечения, но Kafka состоит из огромного количества журналов, распределенных среди множества различных машин. Система связывает их друг с другом, надёжно распределяет сообщения, создает реплики для большей отказоустойчивости и корректно обрабатывает сбои.

И хотя работа на одной машине возможна, кластеры для промышленной эксплуатации, как правило, начинаются от трёх машин, а более крупные насчитывают сотни. Как правило, если вы работаете с темой, будет задействован весь кластер, данные будут распределены среди всех доступных вам машин. Таким образом, масштабирование становится довольно простым делом: добавляйте новые машины и перераспределяйте данные. Потребление также можно выполнять параллельно, распределяя сообщения в теме по нескольким потребителям в группе потребителей (см. рис. 4.2).

С точки зрения архитектуры, основное преимущество подобного подхода в том, что масштабируемость теперь не является проблемой. С Kafka в рамках бизнес-систем, упереться в потолок масштабируемости практически невозможно. Это добавляет уверенности и предоставляет большие возможности, особенно при росте экосистем, позволяя разработчикам выбирать модели, которые требуют большей свободы в плане пропускной способности и движения данных.

Масштабируемость открывает и другие возможности. Отдельные кластеры могут вырасти до масштабов компании без риска повалить инфраструктуру непомерными нагрузками. Например, New Relic базируется на едином кластере[29] из 100 узлов, размещенных в трёх центрах обработки данных и обрабатывающих 30 Гбит/с. В других, менее информационно-емких сферах 5- и 10-узловые кластеры обычно выдерживают нагрузку всей компании.

Но следует отметить, что не все компании идут по пути «одного большого кластера». Netflix, например, советует использовать несколько небольших кластеров[30] для снижения эксплуатационных издержек, связанных с поддержкой очень больших комплексов, хотя их крупнейший кластер до сих пор насчитывает около 200 узлов.

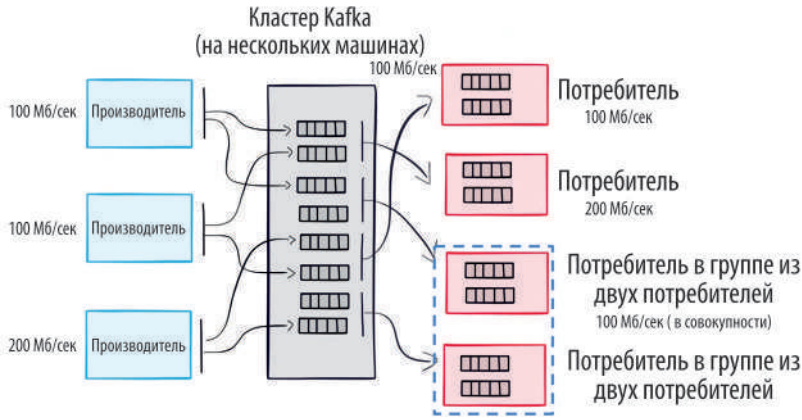


Рис. 4.2. Производители распределяют сообщения по множеству секций на разных машинах, где каждая секция представляет собой маленькую очередь сообщений; выравниваемые по нагрузке потребители (обозначенные как группа потребителей) делят секции между собой; производителям, а также отдельным потребителям и группам потребителей можно ограничить пропускную способность

Для управления распределёнными кластерами полезно разделять пропускную способность сети с помощью опций, предложенных Kafka. Это мы и обсудим далее.

Разделение нагрузки в мультисервисных экосистемах

Сервисные архитектуры по определению подразумевают наличие большого количества пользователей. Один кластер будет использоваться множеством различных сервисов. На самом деле, это не такая уж и редкость, когда все сервисы компании обращаются к одному кластеру в промышленной эксплуатации. Однако такой вариант не сможет застраховать от случайных DoS-атак и последующей деградации или снижения стабильности сервиса.

Чтобы помочь с этим, Kafka предлагает опцию контроля пропускной

способности, называемую квотой,[31] которая позволяет назначить определенный объём трафика для конкретных сервисов, гарантируя, что они функционируют строго в рамках соглашения об уровне обслуживания (англ. service-level agreement, далее — SLA) (см. рис. 4.2). Запросы «жадных» сервисов беспощадно ограничиваются, в результате кластер, используемый несколькими сервисами, застрахован от неожиданного конфликта в сети. Эта опция может применяться как для отдельных экземпляров сервиса, так и для групп балансировки нагрузки.

Выполнение гарантий строгой упорядоченности

Большинству бизнес-систем требуются надёжные гарантии последовательности, хоть это и не часто используется в аналитических случаях. Скажем, клиент несколько раз обновляет свою информацию. Порядок этих изменений очень важен, иначе последние изменения могут быть перезаписаны устаревшими значениями.

Чтобы гарантировать строгий порядок, нужно принять во внимание несколько вещей. Во-первых, сообщения, требующие относительного порядка, должны быть отправлены в одну и ту же секцию. (Kafka предоставляет гарантии порядка только в пределах одной секции.) Это уже сделано за вас, нужно только добавлять один и тот же ключ к сообщениям, требующим относительного порядка. Так что поток обновлений информации клиента будет использовать `CustomerId` в качестве ключа секционирования. Все сообщения, относящиеся к одному и тому же клиенту, будут отправлены в одну секцию, и, следовательно, строго упорядочены (см. рис. 4.3).

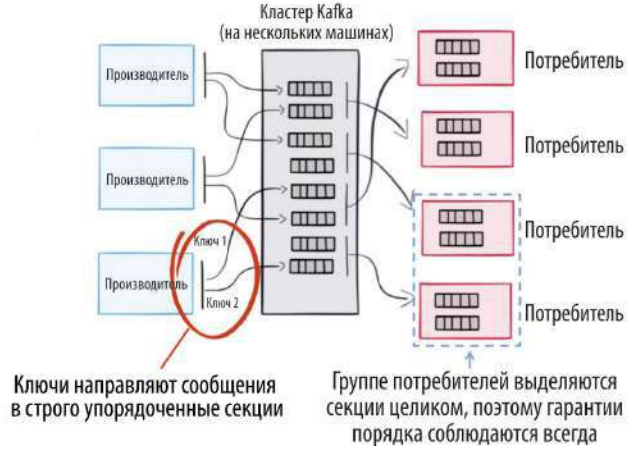


Рис. 4.3. Порядок в Kafka определяется производителем сообщения с помощью ключа последовательности

Иногда упорядоченности на основе ключа может быть недостаточно, и требуется глобальный порядок. Это часто происходит, когда нужно перенести данные более старой системы сообщений, в которой глобальный порядок подразумевается на уровне архитектуры. Чтобы поддерживать глобальный порядок, используйте одну секционную тему. Пропускная способность будет ограничена таковой на одной машине, но обычно для таких ситуаций этого достаточно.

Вторая вещь, о которой следует помнить, это повторные попытки. Почти во всех случаях мы хотим дать возможность повторов в работе производителя, чтобы во время сетевого сбоя, длительного сбора мусора, отказа и т. п. отправку сообщений, которые не удалось успешно доставить в кластер, можно было повторить. Тонкость в том, что сообщения отправляются пакетами, и мы должны быть внимательны и отправлять эти пакеты сообщений по очереди, по одному на машину назначения, чтобы избежать возможности переупорядочивания событий в ситуации, когда при сбое пакеты отправляются повторно. Это легко настроить.[32]

Обеспечение надёжности сообщений

Kafka обеспечивает надежность с помощью репликации. Это означает, что сообщения записываются на настраиваемое количество машин, чтобы при сбое одной или нескольких из них сообщения не были потеряны. Например, если вы устанавливаете коэффициент репликации равный 3, тогда две машины могут выйти из строя без потери данных.

Чтобы наилучшим образом использовать репликацию для таких чувствительных наборов данных как в сервисных приложениях, установите три реплики для каждой секции и настройте производителя так, чтобы он дожидался завершения репликации прежде, чем продолжать работу. И, наконец, как обсуждалось ранее, настройте в производителе возможность повторной отправки.

В отдельных ограниченных случаях может потребоваться, чтобы данные сбрасывались на диск синхронно, но такой подход следует использовать с осторожностью, так как это значительно повлияет на пропускную способность, особенно в высококонкурентных окружениях. Если вы всё же используете этот подход, увеличьте размер пакета производителя, чтобы повысить успешность каждого сброса на диск машины (пакеты сообщений сбрасываются вместе). Этот подход полезен также при развертывании на одной машине, где один узел ZooKeeper запущен на этой же машине, а сообщения сбрасываются на диск синхронно для большей отказоустойчивости.

Балансируем нагрузку сервисов и делаем их высокодоступными

Событийно-ориентированные сервисы должны всегда работать в режиме высокой доступности (англ. high availability). К тому же такая конфигурация не требует дополнительных действий. Если у нас есть один экземпляр сервиса, и мы запускаем второй, то нагрузка будет автоматически распределена по обоим экземплярам. Этот же процесс обеспечивает высокую доступность при падении одного из узлов (см. рис. 4.4).

Скажем, у нас есть два экземпляра сервиса заказов, которые читают сообщения из темы Заказы. Kafka назначит половину секций каждому экземпляру, так что нагрузка распределится по обоим.

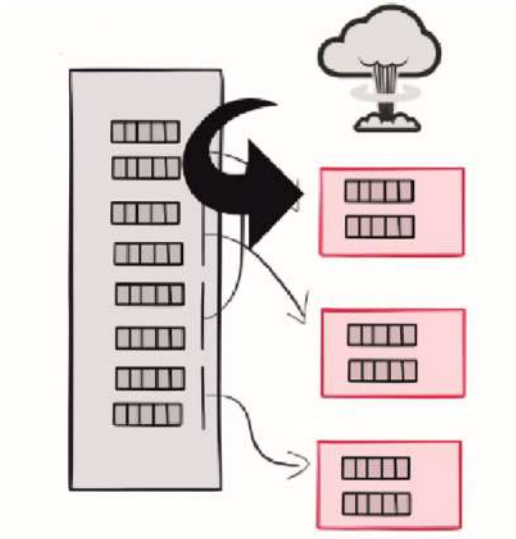


Рис. 4.4. Если экземпляр сервиса умирает, данные перенаправляются, и гарантии упорядоченности сохраняются

При сбое одного из сервисов, Kafka обнаружит этот сбой и перенаправит сообщения к оставшемуся. Если претерпевший сбой сервис снова начнёт работать корректно, нагрузка вернётся обратно.

Этот процесс на самом деле работает путём присвоения целых секций разным потребителям. Сила такого подхода заключается в том, что одна секция может быть назначена только единственному экземпляру сервиса (потребителю). Это неизменно, порядок гарантирован, даже если сервисы терпят сбой или перезапускаются.

Таким образом, сервисы получают и высокую доступность, и балансировку нагрузки, а значит, их можно масштабировать, обрабатывать внеплановые простои или последовательно перезапускаться без простоя (например, для обновления кода). А ещё новые выпуски Kafka всегда обратно-совместимы с предыдущей версией, так что вы гарантированно можете развернуть новую версию без простоя системы.

Уплотнённые темы

По умолчанию темы в Kafka ориентированы на хранение: сообщения хранятся в системе на протяжении некоторого заданного времени. Также в Kafka есть специальный вид тем, управляющий наборами данных со значениями, которые имеют какой-либо ключ. Это данные у которых есть первичный ключ (идентификатор), как в таблице БД. Эти уплотнённые темы (англ. compacted topics) хранят только самые недавние события, удаляя все старые для определённого ключа. Также они поддерживают удаление данных (см. «Удаление данных» на стр. 140 в Главе 13).

Механика работы уплотнённых тем похожа на простое журнально-структурированное дерево со слиянием (англ. log structured merge trees, далее — LSM-дерево).[33] Тема периодически сканируется, и старые сообщения удаляются, если они были заменены сообщениями с таким же ключем, см. рис. 4.5. Стоит отметить, что это асинхронный процесс, так что уплотнённые темы могут содержать заменённые сообщения, ожидающие удаления.

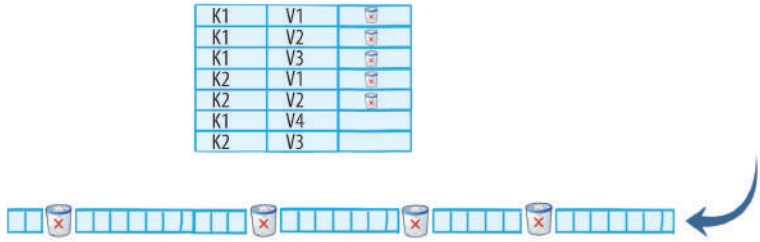


Рис. 4.5. В уплотнённой теме, заменённые сообщения, имеющие один и тот же ключ, удаляются. Так, в этом примере для ключа K2 сообщения V2 и V1 в конечном итоге будут удалены, т. к. они заменяются сообщением V3

Уплотнённые темы позволяют нам применить пару оптимизаций. Первое, они позволяют нам замедлить рост наборов данных (за счёт удаления заменённых событий), но делаем мы это с оглядкой на данные, а не просто бездумно удаляем всё, скажем, старше двух недель. Второе, более маленькие наборы данных проще передавать с машины на машину.

Это важно для контекстно-зависимой потоковой обработки. К примеру, сервис использует API Kafka Streams для загрузки самой свежей версии каталога продуктов в таблицу (как будет обсуждаться в разделе «Оконные функции, соединения, таблицы и хранилище состояний» на стр. 150 в Главе 14, таблица является хранящийся на диске хэш-таблицей внутри API). Если каталог продуктов хранится в Kafka в уплотнённой теме, загрузка может быть проведена быстрее и эффективнее, т. к. не нужно будет передавать всю историю версионирования (в отличие от работы с обычными темами).

Долговременное хранение данных

Одно из самых главных отличий Kafka от других систем обмена сообщениями — возможность применения в качестве прослойки хранилища.[34] На самом деле, нет ничего необычного в том, чтобы обнаружить обычную или уплотнённую тему, содержащую больше 100 Тбайт данных. Но Kafka — это не база данных, это журнал изменений, не предлагающий широкой функциональности запросов (и нет никаких оснований считать, что такое положение дел когда-либо изменится). Но её простой подход оказывается очень полезным для хранения разделённых наборов данных в больших системах или архитектурах — к примеру, для использования событий, как общего источника истины, о чём мы отдельно поговорим в Главе 9.

Данные могут храниться в обычных темах, что отлично подходит для аудита и в качестве источника событий, или в уплотнённых темах, для уменьшения общего размера. Вы можете применять оба варианта, получая преимущества и того, и другого, просто дублируя данные и связывая их вместе с помощью задания Kafka Streams (ценой занимаемого места). Такой подход называется «актуальный-версионированный» шаблон (англ. latest-versioned).

Безопасность

Kafka предоставляет целый ряд функций обеспечения безопасности корпоративного класса как для аутентификации, так и для авторизации. Клиентская аутентификация производится либо с помощью Kerberos, либо с помощью клиентских сертификатов TLS. Таким образом, кластер Kafka всегда точно знает, кто именно делает каждый запрос. Также имеется Unix-подобная система прав, с помощью которой можно управлять тем, какие пользователи к каким данным имеют доступ. Сетевое взаимодействие может быть зашифровано, позволяя безопасно передавать сообщения через недоверенные сети. И, наконец, администраторы могут потребовать

аутентификации для взаимодействия Kafka и ZooKeeper.[35]

Механизм квотирования, который мы обсуждали в разделе «Разделение нагрузки в мультисервисных экосистемах» на стр. 23, может быть связан с этим понятием идентификации, и все функции безопасности Kafka можно будет распространить на различные компоненты платформы Confluent (Rest Proxy, Confluent Schema Registry, Replicator и т. д.).

Заключение

Kafka немного отличается от обычных технологий обмена сообщениями. Будучи спроектирована как распределённый, масштабируемый инструмент, она становится идеальным проводником, с помощью которого сервисы могут сохранять события и обмениваться ими. Безусловно, в ней есть ряд уникальных технологических решений, но выделяют её, в первую очередь, способность масштабироваться, работать без сбоев и долгосрочно хранить наборы данных.

Мы можем использовать шаблоны и возможности, описанные в этой главе, для построения различных архитектур — от небольших сервисных систем до огромных корпоративных конгломератов. Этот подход безопасен, практичен, опробован и оттестирован.

ЧАСТЬ II

Проектирование событийно-ориентированных систем

Жизнь — это последовательность естественных и спонтанных изменений.
Не противьтесь им, ибо это лишь преумножает печаль.
Позвольте реальности быть реальностью.
Дайте вещам идти своим чередом.

— Лао Цзы

ГЛАВА 5

События — основа сотрудничества

Архитектуры на основе сервисов, таких как микросервисы или SOA, обычно строятся на протоколах синхронных запросов и ответов. Это довольно естественный подход. В конце концов, именно так мы и пишем программы: мы вызываем другие модули в коде, ожидаем ответа, а затем идём дальше. Он так же хорошо подходит для множества повседневных ситуаций, например, для веб-сайта, где пользователь нажимает на кнопку и ожидает реакции.

Но в мире множества независимых сервисов подход начинает меняться. С ростом числа сервисов растёт и сеть синхронных взаимодействий. Прежде малозначимые проблемы с доступностью теперь приводят к куда большим простоям. Нашим системным администратора приходится играть в детективов, бегая от сервиса к сервису и по крупицам собирать улики, чтобы раскрыть таинственное распределённое убийство. (Кто что сказал, кому и когда?)

Это хорошо известная проблема, и у неё есть ряд решений. Например, можно обеспечить каждый индивидуальный сервис значительно более высоким SLA, чем у всей системы в целом. Google предоставляет протокол для этого.[36] В качестве альтернативы можно просто разбить синхронные связи, которые соединяют сервисы вместе, с помощью (а) асинхронности и (б) брокера сообщений в качестве посредника.

Допустим, вы работаете в интернет торговле. Вы, вероятно, заметили, что синхронные интерфейсы, такие как `getImage()` или `processOrder()` — вызовы, ожидающие немедленного ответа, воспринимаются естественно и знакомо. Но когда пользователь нажимает кнопку «Купить», происходит активация большого, сложного и асинхронного процесса. Этот процесс приводит к физической доставке покупки к двери покупателя, что выходит далеко за пределы контекста изначального нажатия кнопки.

Поэтому разделение программы на асинхронные потоки позволяет выделять различные задачи, которые необходимо решить, и принять мир, асинхронный по своей природе.

На практике мы склонны принимать это автоматически. Всем нам приходилось опрашивать таблицы базы данных на наличие изменений, или создавать какую-то регулярную задачу в планировщике (англ. cron job), чтобы разбираться с обновлениями. Этими способами можно разорвать узы синхронности, но они всегда воспринимаются как временное решение. И тому есть причина: это действительно временные решения.

Из всего этого можно сделать такой вывод. Императивная модель программирования, в которой мы командуем сервисами, не слишком подходит для ситуаций, где сервисы работают самостоятельно.

В этой главе мы сфокусируемся на оборотной стороне этой архитектурной медали: мы будем компоновать сервисы не через набор команд и запросов, а через поток событий. Это реализация подхода, который используется в индустрии уже много лет, но в то же время он формирует основу для более продвинутых методов, которые мы обсудим в Частях III и V, где смешаем идеи событийно-ориентированной обработки с идеями, которые можно подчерпнуть из работы потоковых платформ.[37]

Команды, события, запросы

Прежде чем мы пойдём дальше, допустим, что существуют три различных способа взаимодействия программ по сети: команды, события и запросы. Если вы не задумывались о таком разделении раньше, то сейчас самое время, поскольку это послужит важным ориентиром в межпроцессных взаимодействиях.

Эти три механизма взаимодействия сервисов можно описать следующим образом (см. таблицу 5-1 и рис. 5.1):

Команды

> Команды — это действия, запросы на операции, которые должны быть выполнены другими сервисами, нечто, призванное изменить состояние системы. Команды выполняются синхронно и обычно сигнализируют о завершении, иногда возвращают и результат выполнения.[i]

i Термин «команда» происходит из принципа CQS (Command Query Separation) за авторством Бертрана Мейера. Здесь мы используем несколько изменённое определение, оставляя опциональным наличие или отсутствие возвращаемого результата. На то есть причина: команда — это запрос чего-то конкретного, что должно произойти в будущем. Иногда желательно не иметь возвращаемого результата, а иногда возвращённое значение является важным. Мартин Фаулер использует пример с выталкиванием из стека, а мы в этой книге используем пример обработки платежей, которая просто возвращает статус успешности выполнения команды.

Пример: processPayment() возвращает информацию о том, успешно ли завершился платёж.

Когда использовать: в операциях, которые должны быть закончены синхронно, либо при использовании оркестрации или диспетчера процессов.[38] Возможно стоит использовать команды только в ограниченном контексте.

События

События — это одновременно и факт, и уведомление. События означают собой, что нечто произошло в реальном мире, но при этом не несут в себе ожидания каких-то дальнейших действий. Они двигаются только в одном направлении и не ожидают ответа (как говорится, «выстрелил и забыл»), но могут быть «произведены» из более поздних событий.

Пример: OrderCreated{Widget}, CustomerDetailsUpdated{Customer}

Когда использовать: когда важны слабые связи (например, в многокомандных системах), где поток событий важен для нескольких сервисов, или где данные должны быть реплицированы из одного приложения в другое. События также поддаются параллельному выполнению.

Запросы

Запросы — это потребность что-то найти. В отличие от событий или команд, запросы свободны от побочных эффектов. Они не влияют на состояние системы.

Пример: getOrder(ID=42) вернёт Orders(42, ...).

Когда использовать: для получения небольших объемов данных от других сервисов, либо для получения большого количества данных внутри сервиса.

Таблица 5-1. Различия между командами, событиями и запросами.

	Поведение/изменение состояния	Включает ответ
Команда	Событие запрошено	Возможно
Событие	Только что произошло	Никогда
Запрос	Ничего	Всегда

Создавая команду с опциональным типом возврата, разработчик может решить, нужно ли возвращать результат или нет, и, если нет — использовать ли CQS/CQRS. Это позволяет обходиться без ещё одного имени для команды, которая ничего не возвращает. И наконец, команда никогда не является событием. Команда явно предполагает, что что-то (какой-то эффект или изменение состояния) обязательно произойдёт в будущем. События происходят без подобных ожиданий. Они являются простым уведомлением о том, что что-то произошло.

Прелесть событий в том, что они одновременно играют две роли: одна заключается в уведомлениях, которые запускают сервис, а вторая — в репликации данных из одного сервиса в другой. Но с точки зрения сервиса, события приводят к меньшей связанности,[39] чем команды и запросы. А слабая связанность — это очень желательный параметр в условиях, когда взаимодействия пересекают границы развёртываний, ведь сервисы с меньшим количеством зависимостей проще изменять.

Рис. 5.1. Визуальное описание команд, событий и запросов

Связанность и брокеры сообщений

Термин «слабая связанность» используется достаточно широко. Изначально это была эвристика для структурирования программ,[40] однако для сетевых сервисов, особенно разрабатываемых разными командами, слабую связанность нужно воспринимать несколько иначе. Вот относительно свежее определение[41] от Фрэнка Лейманна:

> Слабая связь снижает количество допущений, которые делают две стороны друг о друге при обмене информацией.

Допущения эти в целом относятся к комбинации данных, функций и работоспособности. Однако, как выяснилось, это не совсем то, что имеют в виду люди, говоря о слабой связанности. Называя приложения слабосвязанными, чаще подразумевают нечто похожее на «осведомлённость» (фран. connascence), определяя это так:[42]

> Это мера влияния изменений одного компонента на остальные.

Такое определение отражает интуитивное понятие связанности: если две сущности связаны между собой, то действие, направленное на одну из них, каким-то образом отразится на другой. Но в определении слова «connascence» присутствует ключевое слово «изменение», которое говорит

о временности явления. Связанность не статична. Она важна только в тот момент, когда мы вносим изменения в программу. В самом деле, если мы оставим программу в покое и не будем её изменять, степень связанности не будет иметь значения.

Всегда ли хороша слабая связанность?

Широко распространено мнение, что сильная связанность — это всегда плохо, а слабая — всегда хорошо. Это не совсем точно. И сильная, и слабая связанности эффективны в определённых ситуациях. Можно обобщить это следующим образом:

> Слабая связанность позволяет изменять компоненты отдельно друг от друга. Сильная связанность помогает компонентам извлекать больше пользы друг из друга.

Путь к слабой связанности — отказ от общих ресурсов. Если вы ничем не делитесь, другие приложения не могут к вам привязаться. Микросервисы, к примеру, иногда описывают формулой «ничего общего»,[43] стимулируя разные команды не делиться данными и функциями[ii] (за пределами границ сервисов), поскольку это лишает их возможности действовать самостоятельно.[iii]

Конечно, такой подход создаёт проблемы для совместной работы: в конечном итоге, вам неизбежно придётся изобретать велосипед (или заставлять других). И хотя это может быть приемлемо для вас, вы наверняка создадите проблемы для отдела или всей компании. Конечно, разумный подход помогает найти баланс. Большинство бизнес-приложений должны обмениваться информацией друг с другом, поэтому некоторая связанность будет существовать всегда. Общая функциональность сервисов, будь то DNS или обработка платежей, может быть полезна так же, как и библиотеки с общим кодом. Сильная связанность тоже может быть полезной, но это всегда компромисс.

Примечание: Чем бы мы не делились, что-то общее всегда увеличивает связанность.

ii Принцип «ничего общего» часто используется в базах данных, но несёт несколько иной смысл.

iii Байка в тему: однажды я работал с командой, которая шифровала часть публикуемой информации не ради безопасности, а для того, чтобы контролировать, кто может к ней подключаться (выдавая нужным сторонам ключи шифрования). Я бы не рекомендовал такой подход, но эта ситуация показывает, что людей действительно волнует эта проблема.

Например, в большинстве традиционных приложений вы плотно привязаны к базе данных, и приложение будет стараться получить максимум пользы от способности базы данных к выполнению ресурсоёмких операций. Есть и недостаток: поскольку приложение и база данных развиваются вместе, вы предпочтёте не давать другим приложениям доступ к этой базе данных. Другой пример — DNS, широко используемый во всей организации. Его широкое использование делает сервис очень ценным, но одновременно и сильно связанным. Но, поскольку он обладает простым интерфейсом и меняется не часто, в таком положении дел практически нет недостатков.

Мы можем наблюдать, что связанность одного компонента является функцией от трех факторов (с дополнением):

- Зона интерфейса (функциональные возможности, широта и количество предоставляемых данных)
- Количество пользователей
- Операционная стабильность и производительность

Дополнение: частота изменений — если компонент не изменяется (данные, функции или операции), то связанность не имеет значения.

Сообщения помогают строить слабосвязанные сервисы, потому что они перемещают чистые данные из сильно связанных мест (из источника) в слабосвязанные места (к подписчикам). Любая операция с данными производится не в источнике, а на стороне каждого подписчика, а системы передачи сообщений, например Kafka, берут на себя работу по обеспечению производительности и стабильности.

С другой стороны, запросно-ориентированный подход более сильносвязанный, поскольку функциональность, данные и операции сконцентрированы в одном месте. Позже мы обсудим идею ограниченного контекста (англ. bounded context), которая является способом сбалансировать эти два подхода: запросно-ориентированные протоколы используются в ограниченном контексте, а сообщения — между ними. Мы также обсудим более широкие последствия связанности в Главе 8.

Связанность данных неизбежна

У достаточно крупного бизнеса обычно есть ключевые наборы данных, которые используются многими программами. Если вы отправляете пользователю сообщение, вам нужен его адрес; если вы делаете отчёт о продажах, вам нужно количество продаж, и так далее. И если при

недоступности какой-то функциональности можно найти обходной путь, то без данных обойтись уже никак не получится. Так что практически все бизнес-системы в больших организациях нуждаются в необходимом уровне связанности данных.

Примечание: Функциональная связанность не является обязательной. А связанность ключевых данных неизбежна.

Использование событий для уведомлений

Большинство брокеров сообщений предоставляют систему издателей-подписчиков, где логика доставки сообщений определяется не отправителем, а получателем. Этот процесс известен как маршрутизация на стороне получателя (англ. receiver-driven routing). Получатель контролирует своё присутствие во взаимодействии, что обеспечивает систему модульностью (см. рис. 5.2).

Рис. 5.2. Сравнение подхода «запрос-ответ» и событийно-ориентированного подхода для демонстрации меньшей связанности последнего

Рассмотрим простой пример, в котором клиент заказывает iPad. Пользователь нажимает кнопку «Купить», и заказ немедленно отправляется в сервис заказов. Происходят три вещи:

1. Сервис доставки уведомлён.
2. Он ищет адрес для отправки iPad.
3. Он начинает процесс отправки.

В подходе на основе REST или RPC это может выглядеть как на рис. 5.3.

Рис. 5.3. Система управления заказами, основанная на запросах

Тот же рабочий процесс может быть построен в рамках событийно-ориентированного подхода (рис. 5.4), где сервис заказов просто записывает событие «Заказ создан», на которое реагирует сервис доставки.

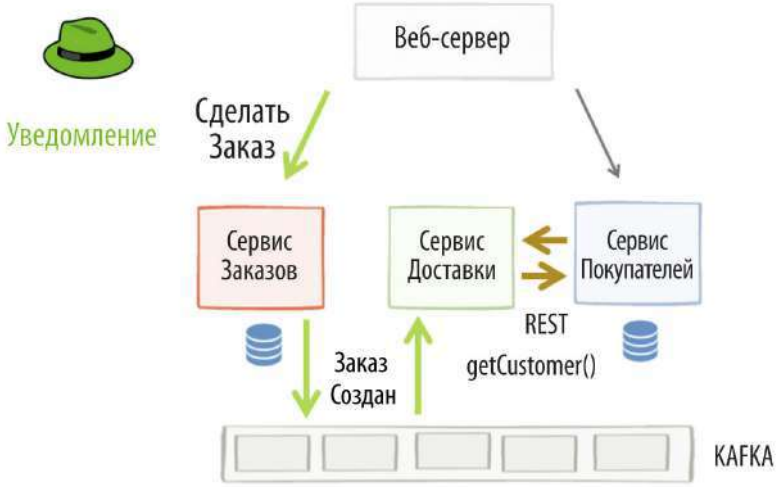


Рис. 5.4. Событийно-ориентированная версия системы из рис. 5.3; в этой конфигурации события используются лишь в качестве уведомлений: сервис заказов уведомляет сервис доставки через Kafka

Если приглядеться, взаимодействие между сервисами заказов и доставки не сильно изменилось, они просто стали общаться через события, а не через прямые вызовы. Однако важное изменение всё-таки произошло: сервис заказов теперь не знает о том, что сервис доставки вообще существует.

Он просто вызывает событие, которое говорит, что он выполнил работу, и заказ был создан. Сервис доставки теперь сам решает, будет ли он в этом участвовать. Это пример маршрутизации, управляемой получателем: логика теперь лежит на получателе события, а не на отправителе. Бремя ответственности переложено на другие плечи! Это снижает связанность и добавляет системе модульности.

С ростом сложности системы модульность приобретает большое значение. Скажем, мы захотим расширить систему, добавив сервис переоценки, который обновляет цену товара в реальном времени, изменяя её в зависимости от спроса и предложения (рис. 5.5). В подходах на REST и RPC нам понадобится метод вроде `maybeUpdatePrice()`, который будет вызывать и сервис заказов, и сервис доставки. А в событийно-ориентированной модели сервис переоценки просто встраивается в поток событий заказов и платежей, отправляя обновления цены при наступлении соответствующих условий.

Рис. 5.5. Расширение системы из рис. 5.4 путём добавления сервиса переоценки демонстрирует модульность архитектуры

Использование событий для передачи состояния

На рис. 5.5 мы использовали события в качестве уведомлений, но оставили запрос адреса клиента в виде REST/RPC вызова.

Мы могли бы использовать события в качестве типа передачи состояния, чтобы вместо отправки запроса к сервису клиентов использовать поток событий для репликации информации о клиенте из сервиса клиента в сервис доставки, где он мог бы быть запрошен локально (см. рис. 5.6).

Это позволяет использовать еще одну особенность событий — возможность репликации данных. (Формально это называется событийной

Рис. 5.6. Расширение системы из рис. 5.4 до полностью событийно-ориентированной; здесь события используются для уведомлений (сервис заказа уведомляет сервис доставки), а также для репликации данных (данные передаются из сервиса клиента в сервис доставки, где могут быть запрошены локально)

передачей состояния (англ. event-carried state transfer),[44] что является одной из форм интеграции данных.[45]) Уведомления делают архитектуру более модульной, а репликация позволяет передавать данные между сервисами для дальнейших локальных запросов. Репликация наборов данных на локальный уровень выгодна по тем же причинам, по которым выгоден кэш — она ускоряет доступ к данным.

Какой подход использовать

Можно обобщить преимущества подхода «запрос через событийную передачу состояний» следующим образом:

Большая изоляция и автономность
: Изоляция требуется для автономности. Хранение данных для локальных изолированных запросов означает, что данные остаются под контролем сервиса.

Более быстрый доступ к данным
: Доступ к локальным данным обычно происходит быстрее. Это особенно важно, когда нужно объединить данные из нескольких источников, или когда запрос охватывает разные регионы.

Необходимость автономного доступа к данным
: В случае с мобильными устройствами или в транспортных средствах (на самолёте, корабле, и так далее) репликация наборов данных

предоставляет механизм для непрерывной работы и синхронизации при подключении.

С другой стороны, есть преимущества и у подходов REST/RPC:

Простота
: Это проще реализовать, потому что в системе имеется меньше элементов и нет состояния, которым нужно управлять.

Единое состояние
: Состояние находится только в одном месте (помимо кэша), а это значит, что изменение значения будет немедленно видно всем пользователям. Это важно в случаях, когда требуется синхронность, например, для чтения предыдущего обновления баланса на банковском счете (мы посмотрим, как наделить такой особенностью события в разделе «Нарушение принципа CQRS через блокировку чтения» на стр. 156 в Главе 15).

Централизованное управление
: Системы управления и контроля рабочих процессов могут быть использованы для централизации бизнес-процессов в едином контролирующем сервисе. Это делает систему более понятной и предсказуемой.

Конечно, как мы уже видели, можно смешивать два подхода, и, усиливая один из них, получать решение для архитектур разных размеров. Если мы проектируем небольшую и лёгкую систему, например, онлайн приложение, можно сконцентрироваться на уведомлениях, поскольку репликация данных может показаться лишней надстройкой. Но в более крупных и сложных системах мы можем уделить больше внимания репликации, чтобы каждый сервис был автономным в отношении данных, которые он опрашивает. (Это обсуждается более подробно в Главе 8.) Микросервисные приложения обычно более крупные и используют оба подхода. Джонас Бонер говорит об этом прямо:[46]

> Общение между микросервисами должно основываться на асинхронной передаче сообщений, а логика внутри каждого микросервиса должна работать в синхронном режиме.

Нужно помнить, что он говорит о строгом определении микросервисов, по которому микросервисы разворачиваются независимо. В более нерадивой интерпретации, весьма распространённой, это не является таким строгим условием.

Шаблон «событийное сотрудничество»

Для создания элегантных событийно-ориентированных сервисов зачастую используется шаблон под названием «событийное сотрудничество». Это позволяет набору сервисов совместно работать вокруг одного бизнес-процесса, где у каждого из сервисов есть небольшая задача: слушать события и создавать новые. Так, например, мы можем начать с создания заказа, и сервисы будут продолжать этот процесс, пока заказ не достигнет покупателя.
Может показаться, что это то же самое, что и любой другой схожий подход, но ключевое отличие этого шаблона в том, что ни один из сервисов единолично не управляет всем процессом. Вместо этого каждый сервис отвечает только за небольшую часть — некоторое подмножество переходных состояний, и они соединяются друг с другом через цепочку событий. Каждый сервис выполняет свою работу, а затем создает событие, сообщающее о том, что он сделал. Если он обрабатывает платёж, он создаст событие «Платёж обработан». Если он подтверждает заказ, он создаст событие «Заказ подтверждён», и так далее. Эти события запускают следующий шаг в цепочке (который может снова вызвать этот сервис или активировать другой сервис).

На рис. 5.7 каждый круг представляет собой событие. Цвет круга обозначает тему, к которой он относится. Процесс идет от «Заказ запрошен» до «Заказ выполнен». Эти три сервиса (заказ, платёж, доставка) управляют только теми переходами состояний, которые относятся к их части процесса. Важно, что сервисы не знают о существовании друг друга, и ни один из сервисов не управляет всем процессом. К примеру, платёжный сервис знает лишь то, что он должен среагировать на подтверждённый заказ и создать событие «Платёж обработан», что продвинет процесс обработки заказа на еще один шаг вперед. Таким образом, «валютой» событийного сотрудничества являются, как не удивительно, события!

Благодаря отсутствию единой точки контроля подобные системы называют хореографией: каждый сервис обрабатывает некоторое подмножество переходных состояний, которые, если сложить их вместе, будут описывать весь процесс. Это можно противопоставить оркестрации, где единый сервис управляет и командует всем бизнес-процессом из одного места, например, через диспетчер процессов.[47] Диспетчер процессов реализован через «запрос-ответ».

Хореографические системы имеют преимущество в том, что являются модульными. Если платёжный сервис решит, что надо создать три новых типа событий для платёжной части общего процесса, то он может делать

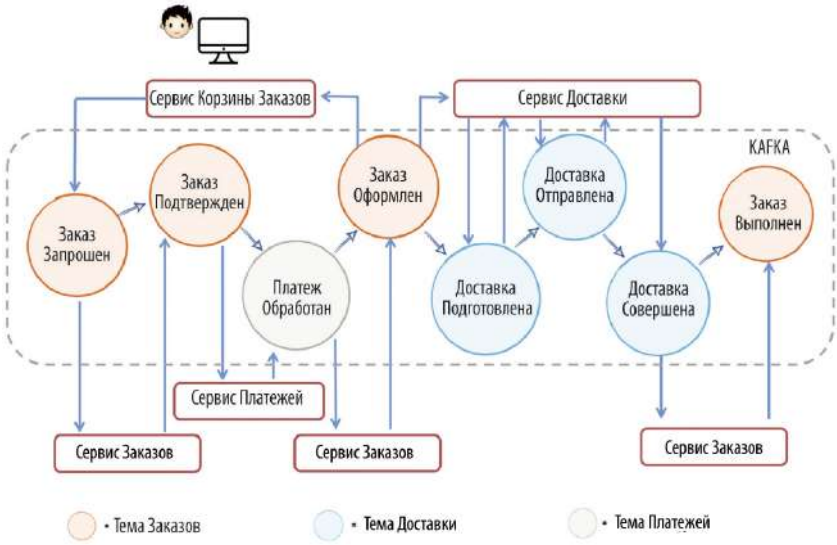


Рис. 5.7. Пример процесса, реализованного с помощью событийного сотрудничества

это безболезненно до тех пор, пока событие «Платёж обработан» остаётся таким же. Это удобно, потому что означает, что, когда вы реализуете сервис, то можете менять способ его работы, и другие сервисы не будут задеты — им вообще не обязательно об этом знать. В случае с оркестрацией, единый сервис диктует работу всему рабочему процессу, и все изменения должны быть сделаны на уровне контроллера. Какой из подходов лучше для вас, будет зависеть от ситуации, но к неоспоримым преимуществам оркестрации можно отнести то, что весь процесс зафиксирован в коде, централизованно. Это позволяет легко рассуждать о системе. Недостатком же является тесная связь системы с контроллером, так что в целом хореография лучше подходит для более крупных реализаций (особенно тех, над которыми работают разные команды, и где сервисы изменяются независимо).

Примечание: События, которыми делятся сервисы, формируют журнал, или «общее повествование», которое описывает, как развивается бизнес с течением времени.

Взаимодействие с потоковой обработкой

Двойственность уведомлений и репликации, которую показывают события, приводят к концепциям контекстно-независимой и контекстно-зависимой потоковой обработки соответственно. Лучше всего это можно

понять через пример с сервисом доставки, который мы обсуждали ранее. Если перевести сервис доставки на Kafka Streams API, то мы сможем подойти к проблеме с двух сторон (рис. 5.8):

Контекстно-зависимый подход

Реплицируем таблицу с клиентами в Kafka Streams API (обозначается как «KTable» на рис. 5.8). Это обеспечит подход с передачей состояний через события.

Контекстно-независимый подход

Мы обрабатываем события и ищем нужного клиента с каждым обработанным заказом.

Рис. 5.8. Контекстно-зависимая потоковая обработка походит на использование событий для уведомлений и передачи состояния (слева), а Контекстно-независимая потоковая обработка похожа на использование событий только для уведомлений (справа)

По сравнению с более ранним примером из этой главы передача состояний в контекстно-зависимой потоковой обработке отличается в двух важных аспектах:

- Набор данных должен полностью храниться в Kafka. Если мы присоединяемся к таблице клиентов, все записи клиентов должны храниться в Kafka в виде событий.
- Обработчик потока включает в себя внутрипроцессное дисковое хранилище для размещения таблицы. Отсутствие внешней базы данных делает этот сервис контекстно-зависимым. Затем Kafka Streams применяет набор методов, чтобы управлять подобным сервисом было практично.

Эта тема обсуждается подробно в Главе 6.

Смешение подходов: событийно-ориентированный и «запрос-ответ»

Общий подход, который часто можно встретить в небольших веб-ориентированных системах, заключается в смешении подходов, как показано на рис. 5.9. Онлайн-сервис взаимодействует напрямую с пользователем, например, через REST, но в то же время записывает изменения состояний в Kafka (см. «Коротко про источники событий, источники команд и CQRS» на стр. 61 в Главе 7). Оффлайновые сервисы (для платежей, исполнений обязательств и т. п.) построены исключительно на событиях.

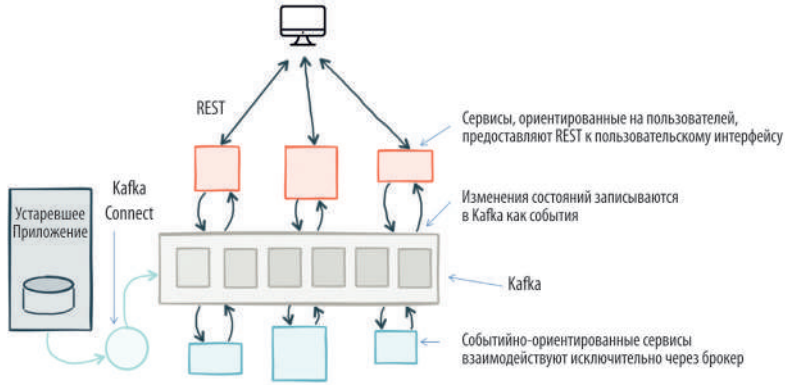


Рис. 5.9. Простой пример событийно-ориентированного сервиса, где данные импортируются из устаревшего приложения через Connect API; сервисы, ориентированные на пользователей, предоставляют REST API к визуальному интерфейсу; изменения состояний записываются в Kafka как события. Внизу — бизнес-процессы выполняются через событийное сотрудничество

В крупных реализациях сервисы группируются вместе, например, по отделам или командам. Они совмещают подходы внутри одного кластера, но полагаются на события для общения между самими кластерами (см. рис. 5.10).

На рис. 5.10 три отдела общаются друг с другом через события. В каждом отделе (три больших круга) интерфейсы сервисов доступны более свободно и используют более отточенный событийно-ориентированный поток, который ведет взаимодействие. Каждый отдел содержит набор внутренних ограниченных контекстов — небольших групп сервисов, которые объединяет предметная область, которые развёрнуты вместе и плотно взаимодействуют друг с другом. На практике, часто возникает определённая иерархия для взаимодействия обмена данными. На

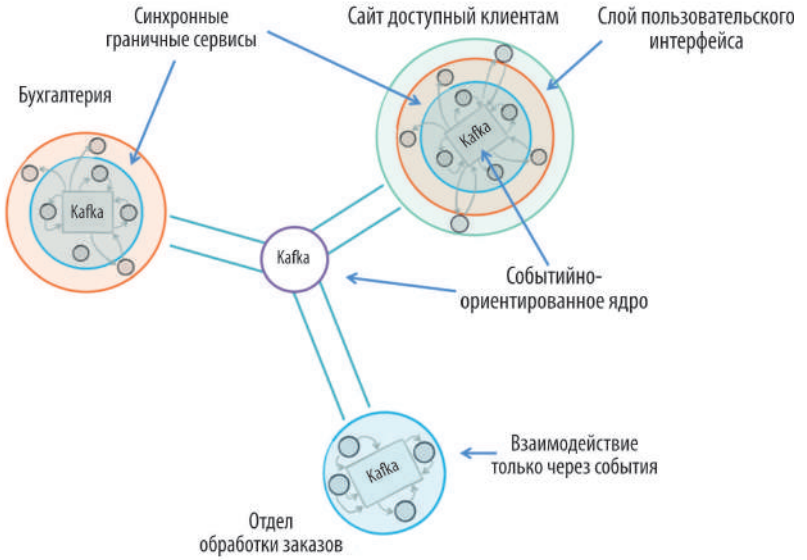


Рис. 5.10. Кластеры сервисов формируют ограниченные контексты с общей функциональностью внутри. Контексты взаимодействуют друг с другом только через события, охватывающие отделы, регионы или облака

верхнем уровне отделы связаны слабо: они делятся между собой только событиями. Внутри отдела находится множество приложений, которые взаимодействуют друг с другом и путём запросов-ответов, и с помощью событийно-ориентированной системы, как на рис. 5.9. Несколько сервисов могут составлять одно приложение, но в целом они будут сильно связаны друг с другом, разделяя предметную область и обладая синхронизированными расписаниями выпуска.

Этот подход, который ограничивает повторное использование в пределах ограниченного контекста, пришёл из предметно-ориентированного проектирования (англ. domain-driven design, далее – DDD).[48] Одна из важных идей DDD состояла в том, что широкое повторное использование может быть неэффективным, и наилучшим подходом является создание границ вокруг бизнес-области и раздельное формирование этих предметных областей. В пределах ограниченного контекста, предметная область является общей, и всё доступно всем, но другие ограниченные контексты, не разделяющие ту же предметную область, обычно взаимодействуют через более ограниченные интерфейсы.

Эта идея получила развитие у тех, кто внедряет микросервисы. Таким образом, ограниченный контекст описывается как набор родственных компонентов или сервисов, разделяющих кодовую базу

и разворачивающихся вместе. Между ограниченными контекстами происходит меньше обмена (кода, функций или данных). На самом деле, как мы уже заметили в этой главе, микросервисы часто описывают как «ничего общего» именно по этой причине.

Заключение

Бизнес — это совокупность людей, команд и отделов, которые выполняют широкий спектр функций, опираясь на технологии. Чтобы быть эффективными, команды должны работать асинхронно по отношению друг к другу, и многие бизнес-процессы тоже асинхронны, например, доставка посылки получателю. Мы можем начать проект в виде веб-сайта, где пользовательский интерфейс делает синхронные вызовы к бизнес-логике, но с ростом это приведёт к тому, что синхронные вызовы плотно свяжут сервисы во время выполнения. Событийно-ориентированные методы действуют иначе, разделяя системы во времени и позволяя им развиваться независимо друг от друга.

В этой главе мы уже заметили, что события, по сути, обладают двумя независимыми ролями: одна нужна для уведомлений (призыв к действию), а вторая нужна для механизма передачи состояний (отправка данных туда, где это необходимо). События делают систему модульной, и для архитектур разумных размеров целесообразно сочетать подходы на основе событий и на основе запросов. Но следует быть осторожным, прибегая к подобному дуализму: каждый из подходов приведёт к очень разным архитектурным решениям. Наконец, мы рассмотрели, как масштабировать оба подхода путём разделения различных ограниченных контекстов, которые взаимодействуют с помощью событий.

Но, на фоне всех этих разговоров о событиях мы мало касались воспроизводимых журналов или потоковой обработки. Применяя эти шаблоны с помощью Kafka, сам используемый инструментарий открывает нам новые возможности. Возможность хранения в брокере сама по себе становится инструментом, под который мы можем проектировать. Это позволяет принять концепцию внешних данных (англ. data on the outside)[49] с центральным источником событий, к которому может обращаться сервис. Системные администраторы, о которых мы говорили во вступительном разделе этой главы, всё-таки будут играть детективов, но, надеюсь, не так часто, и теперь у них в руках хотя бы будет сценарий!

ГЛАВА 6

Обработка событий с помощью функций состояния

Императивные стили программирования являются одними из старейших, и они не даром пользуются популярностью до сих пор. Процедуры выполняются последовательно, словно рассказывают историю, соответственно меняя состояние программы в процессе этого рассказа.

В 80-х и 90-х, когда приложения становились распределёнными, они сохранили тот же самый подход. Подходы как у Corba и EJB (англ. Enterprise JavaBeans) подняли уровень абстракции, сделав распределённое программирование более доступным. Впрочем, история не всегда судит справедливо. Казавшийся в своё время панацеей, EJB быстро потерял популярность из-за сильной связанности и заблуждения, что сеть должна быть абстрагирована от программиста.

Честно признаемся, с тех пор ситуация улучшилась благодаря технологиям вроде gRPC[50] и Finagle,[51] которые добавили элементы асинхронности в запросно-ориентированный стиль. Но такой подход к проектированию распределённых систем не обязательно будет самым продуктивным или устойчивым. Два стиля программирования, которые лучше подходят для распределённой архитектуры, основаны на стиле программирования потоков данных и на функциональном стиле.

Вы столкнётесь с программированием потоков данных, если будете работать с инструментами вроде Sed или с языком Awk.[52] Они в основном используются для обработки текста; например, поток строк может быть пропущен через регулярные выражения, строка за строкой, а на выходе они отправляются в следующую инструкцию, соединяясь через stdin и stdout. Такой стиль программирования похож на сборочный конвейер, где каждый работник выполняет определённое действие над изделием по мере движения ленты. Поскольку каждый из работников сосредоточен

только на наличии входящих предметов (данных), то не существует никаких «скрытых состояний», которые стоило бы отслеживать. Это очень похоже на то, как работают потоковые системы. События накапливаются в потоковом обработчике и ожидают выполнения определённых условий, например, операции объединения двух потоков. Когда нужные события присутствуют, операция объединения завершается, и процесс переключается на следующую команду. Таким образом, Kafka представляет собой эквивалент конвейера (англ. pipe) в оболочке Unix, а обработчики потока обеспечивают организацию цепочки функций.[53]

Присутствует некоторая аналогия с функциональным программированием. Как и со стилем обработки потоков данных, состояния не изменяются в конкретной функции, а развиваются от функции к функции, и это в чём-то совпадает с потоковой обработкой.[54] Поэтому многие преимущества и функциональных языков, и методов программирования потоков данных так же присутствуют и в потоковых системах. Их можно обобщить следующим образом:

- Потоковой обработке присущи способности параллелизации.
- Создание кэшированных наборов данных и поддержание их в актуальном состоянии является естественной частью работы с потоками. Это делает потоковый подход удобным в системах, где данные и код разделены сетью, особенно при обработке данных и выводе графического интерфейса.
- Потоковые системы более устойчивы по сравнению с традиционными подходами, поскольку высокая доступность является важной частью этой архитектуры, и при выполнении данные не теряются (см. обсуждение этой особенности источника событий в Главе 7).
- Потоковые функции обычно делают программы более понятными и сбалансированными. В чистых (англ. pure) функциях побочных эффектов нет. Однако, побочные эффекты есть в обычных функциях, но это обходится отказом от использования разделяемой памяти.
- Потоковые системы открыты для разнообразных технологий, будь то разные языки программирования или разные хранилища данных.
- Программы пишутся на более высоком уровне абстракции, что делает их более понятными.

Но потоковый подход также наследует и некоторые недостатки. Чистые функциональные языки (англ. pure functional languages)[55] должны улаживать несоответствия при взаимодействии с сущностями с состоянием, такими как файловая система или сеть. В подобном ключе

потоковые системы должны переводить свои операции в запросно-ответный стиль REST или RPC и обратно. Это приводит к тому, что иногда вокруг функционального ядра[56] выстраивают специальные системы, которые обрабатывают события асинхронно, оборачивают их в императивную оболочку для обмена с внешними запросно-ответными интерфейсами. Шаблон «функциональное ядро, императивная оболочка» содержит в себе элементы гибкой и масштабируемой системы, поощряя сервисы избегать побочные эффекты, выражая бизнес-логику в простых функциях, соединённых последовательно через журнал.

В следующем разделе мы более детально рассмотрим, почему контекстозависимость очень важна в потоковой обработке.

Делаем сервисы контекстно-зависимыми

Существует популярная мантра, что «контекстонезависимость — это всегда хорошо», и тому есть причины. Контекстно-независимый сервис запускается мгновенно (не требуется загрузка данных) и может быть масштабирован горизонтально простым тиражированием.

Хорошим примером являются веб-серверы: чтобы отдавать больше динамического контента, мы можем масштабировать веб-уровень горизонтально, просто добавляя новые серверы. Так зачем нам нужно что-то ещё? Загвоздка в том, что большинство приложений на самом деле не являются контекстно-независимыми. Веб-серверу нужно знать, какие страницы отрисовывать, какие сессии активны и так далее. Он решает эти проблемы путём хранения состояний в базе данных. Поэтому база данных является контекстно-зависимой, а сервер — контекстно-независимым. Проблема состояния опустилась на уровень ниже. Но с увеличением нагрузки на сайт программистам приходится реализовывать локальное кэширование, что ведет к вечным проблемам с когерентностью кэша и перебором стратегий инвалидации.

У потоковых платформ иной подход к вопросу, где должно храниться состояние. Во-первых, напомним, что события являются фактами, сходящимися к обработчику потока, как конвейерные ленты на сборочной линии. Так, во многих случаях, события, запускающие процесс, содержат все необходимые для программы данные, как и программы с потоками данных, которые мы обсуждали ранее. Если вам нужно проверить содержимое заказа, вам понадобится лишь поток событий.

Иногда этот стиль контекстно-независимой обработки получается естественным образом, а иногда события дополняют заранее,[57] чтобы они точно содержали все данные, нужные для выполнения действия. Но дополнение требует использования поиска, и, как правило, в базе данных.

Подсистемы контекстно-зависимой потоковой обработки, вроде Kafka Streams API, идут ещё дальше: они обеспечивают проведение всех вычислений, которые загружаются в API, заранее, будь то события или таблицы, по которым нужно произвести поиск, или дополнение. Во многих случаях это делает API, и, следовательно, приложение контекстно-зависимыми, и, если по какой-то причине они перезагружаются, для продолжения работы потребуется вернуть состояние.

Возможно, это выглядит бессмысленно. Зачем же тогда делать контекстно-зависимый сервис? Посмотрите на это как на продвинутый способ кэширования, подходящий для ресурсоёмких задач с данными. Давайте посмотрим на три примера — с поиском по базе данных, с контекстно-зависимым и контекстно-независимым событийно-ориентированным подходом.

Событийно-ориентированный подход

Допустим, у нас есть сервис электронной почты, который принимает поток событий заказов и отправляет письмо-подтверждение покупателю в тот момент, когда покупка была совершена. Для этого потребуется информация и о заказе, и о соответствующем платеже. Такой сервис можно создать разными способами. Давайте представим, что это простой событийно-ориентированный сервис (т. е. без использования Streaming API, как на рис. 6.1). Он может реагировать на событие заказа, а затем искать соответствующий платёж. Или наоборот, реагировать на платёж и искать соответствующий заказ. Для примера возьмём первый случай.

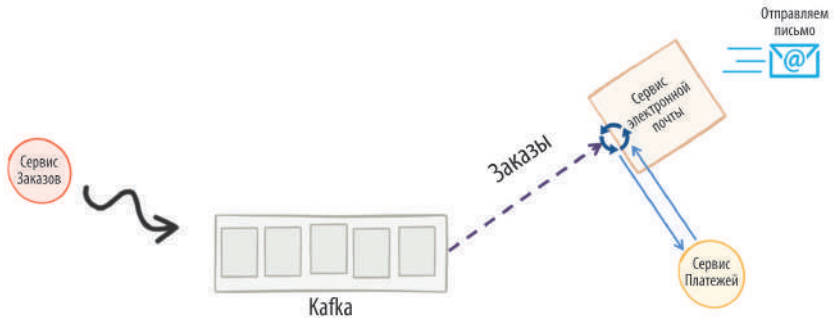


Рис. 6.1. Простой событийно-ориентированный сервис, который ищет необходимые данные по мере обработки сообщений

Обрабатывается единственный поток событий, внутри которого происходит поиск необходимых данных. У такого решения есть две проблемы:

- Требуется постоянно искать данные, по одному сообщению за раз.
- Платежи и заказы создаются примерно в одно и то же время, поэтому платёж может прийти раньше заказа. Это означает, что если заказ попадёт в сервис электронной почты до того, как платёж станет доступен в базе данных, то нам либо придётся заморозить операцию и опрашивать базу, пока платёж там не появится, либо, ещё хуже, вообще пропустить создание электронного письма.

Чистый (контекстно-независимый) потоковый подход

Потоковая система подходит к этой проблеме с другой стороны. Потоки буферизуются, пока не поступят оба события, а затем могут объединиться (рис. 6.2).

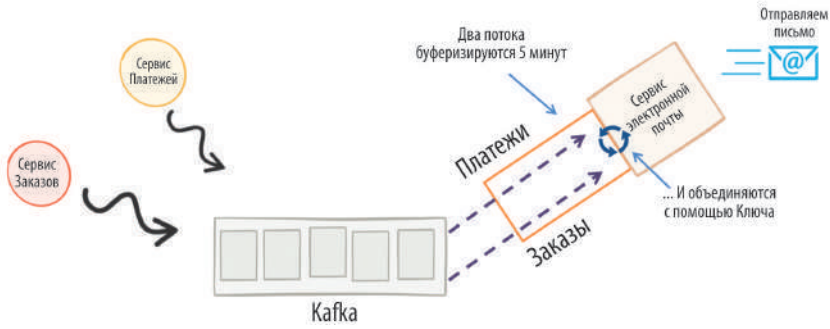


Рис. 6.2. Контекстно-независимый потоковый сервис, объединяющий два потока во время выполнения

Это решает вышеупомянутую проблему событийно-ориентированного подхода. Удалённый поиск теперь отсутствует, решая первую проблему. И не важно, в какой последовательности поступили события, что решает вторую проблему.

И это очень важный момент. При работе с асинхронными каналами нет простого способа удостовериться в их последовательности относительно друг друга. Даже если мы знаем, что заказ всегда создаётся раньше, чем платёж, у заказа может возникнуть задержка, и он прибудет позже.

Наконец, такой подход, строго говоря, не является контекстно-независимым. Буфер делает сервис электронной почты в некотором смысле контекстно-зависимым. При перезапуске Kafka Streams прежде всего происходит загрузка содержимого каждого буфера. Это важно для достижения детерминированного результата. К примеру, вывод операции

объединения зависит от содержимого противоположенного буфера при поступлении сообщения.

Контекстно-зависимый потоковый подход

Увы, данных, поступающих через различные потоки событий, бывает недостаточно, иногда приходится прибегать к поиску или дополнению. К примеру, сервису электронной почты нужен доступ к адресу покупателя. У адреса не происходило недавних событий (конечно, если по счастливой случайности клиент не обновлял свой адрес как раз в это время). Поэтому адрес придётся искать через сервис покупателей, например, с помощью вызова REST (рис. 6.3).

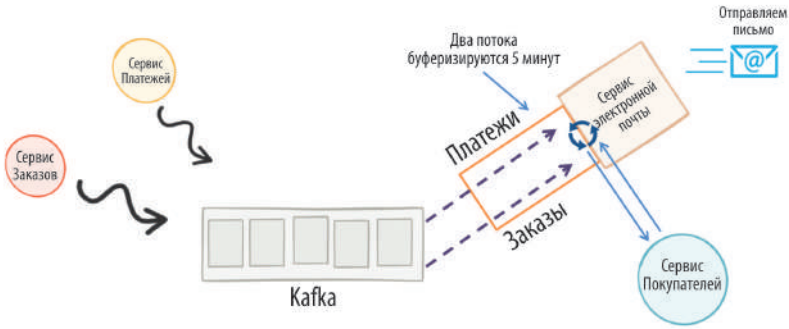


Рис. 6.3. Контекстно-независимый сервис, который ищет данные в другом сервисе при выполнении

Это совершенно нормальный подход (и многие системы работают именно так), но контекстно-зависимая потоковая обработка может его оптимизировать. Используя тот же процесс локальной буферизации, который помог с потенциальными задержками, можно предзагружать весь поток событий о покупателе из Kafka прямо в сервис электронной почты, где и производить поиск исторических значений (рис. 6.4).

Теперь сервис электронной почты не только буферизирует недавние события, но и производит локальный поиск по таблице. (Этот механизм обсуждается подробнее в Главе 14.)

Этот последний, контекстно-зависимый подход, имеет недостатки:

- Сервис теперь контекстно-зависимый. Это означает, что для работы экземпляра почтового сервиса ему потребуются соответствующие данные о клиенте. В худшем случае это приведёт к загрузке полного набора данных при запуске.

Но есть и преимущества:

- Теперь сервис не зависит от недоступности или падения производительности сервиса клиентов.
- Сервис может обрабатывать события быстрее, поскольку каждая операция производится без сетевого запроса.
- Сервис может выполнять больше операций, ориентированных на данные, которые он содержит.

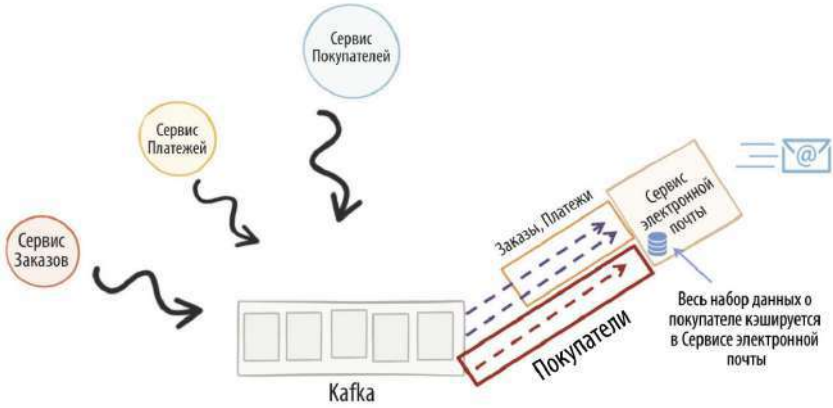


Рис. 6.4. Контекстно-зависимый потоковый сервис, реплицирующий тему Покупатели в локальную таблицу, которая удерживается в Kafka Streams API

Последний пункт очень важен для современных систем, которые всё чаще строятся на работе с большим объёмом данных. В качестве примера представьте графический интерфейс пользователя (англ. graphical user interface, далее — UI или GUI), позволяющий пользователям смотреть заказы, платежи и свою информацию в виде прокручивающейся сетки. Сетка может прокручиваться вверх и вниз, отображая различные данные.

Рис. 6.5. Контекстно-зависимая потоковая обработка используется для материализации данных внутри веб-сервера, чтобы UI мог быстро к ним обращаться, в данном случае — через прокручивающуюся сетку

В традиционной контекстно-независимой модели каждая строка на экране потребует вызова всех трёх сервисов. Это не самый быстрый подход, поэтому придётся добавлять кэширование, а также придумать механизмы для поддержания кэша в актуальном состоянии.

А при потоковой реализации данные будут отправляться в UI непрерывно (рис. 6.5). Поэтому вы сможете определить запрос данных для отображения в сетке, что-то вроде `select * from orders, payments, customers where…` API выполнит его по входящим потокам событий, сохранит результат локально и будет поддерживать его в актуальном состоянии. Поток в этом случае является чем-то вроде декларативно объявленного кэша.

Практические вопросы контекстозависимости

Контекстозависимость сопровождается определёнными трудностями: все контекстно-зависимые компоненты (т. е. хранилища состояний) должны быть загружены при запуске до того, как можно будет начать обработку сообщений, и в худшем случае это может занять продолжительное время. Kafka Streams предлагает три механизма для решения этой проблемы, которые делают контекстозависимость более практичной:

- Используется техника под названием «реплика ожидания» (англ. standby replica),[58] которая гарантирует, что на каждую таблицу или хранилище состояний, хранимых на одном узле, приходится актуальная реплика, хранимая на другом. Таким образом, если узел выйдет из строя, его моментально заменит запасной узел без прерывания обработки.
- Периодически создаются контрольные точки на дисках,[59] чтобы в случае проблем и перезапуске узла он мог восстановиться из контрольной точки, а из журнала получить только пропущенные, за время простоя, сообщения.
- Наконец, уплотнённые темы[60] нужны для того, чтобы набор данных не становился слишком большим. Благодаря этому снижается время загрузки для полного восстановления, если это когда-нибудь понадобится.

Примечание:

Событийно-ориентированное приложение использует один входящий поток для управления работой. Потоковые приложения смешивают один или несколько входящих потоков в один или несколько исходящих потоков. Контекстно-зависимое потоковое приложение также превращает потоки в таблицы (для дополнения) и сохраняет промежуточные состояния в журнал, таким образом содержит в себе все необходимые данные.

Kafka Streams используют промежуточные темы, которые могут быть сброшены или повторно извлечены с помощью инструмента Streams Reset (сброс потоков).[61]

Заключение

В этой главе мы рассмотрели три подхода к обработке на основе событий: простой событийно-ориентированный подход, где вы обрабатываете единственный поток, по одному сообщению за раз; потоковый подход, где вы объединяете несколько потоков вместе; и контекстно-зависимый подход, где вы превращаете потоки в таблицы и сохраняете данные в журнал.

Вместо спускания проблемы с состоянием на уровень баз данных, контекстно-зависимые обработчики, такие как Kafka Streams API, сами хранят контекст. Они делают данные доступными там, где это необходимо. Это увеличивает автономность и производительность. Удалённые вызовы не нужны!

Разумеется, у контекстозависимости есть и недостатки, но она необязательна. К тому же в реальном мире потоковые системы часто объединяют в себе все три подхода. Мы обсудим подробности таких потоковых операций в Главе 14.

ГЛАВА 7

Источники событий, CQRS и другие модели состояний

В Главе 5 мы ввели шаблон «событийное сотрудничество», когда события описывают развивающийся бизнес-процесс, например, обработку покупки или бронирование и расчёт сделки: несколько сервисов работают вместе, чтобы достигнуть результата.

Это ведёт к появлению журнала с постоянной записью о каждом изменении состояния, которое происходит в системе, что, в свою очередь, приводит к появлению двух связанных шаблонов — «командно-запросная изоляция» (англ. command query response segregation, далее — CQRS)[62] и «источники событий» (англ. event sourcing).[63] Эти шаблоны помогают системе быть масштабируемой и стойкой к повреждениям. В этой главе мы раскроем значения этих концепций, как и где их следует применять.

Коротко про источники событий, источники команд и CQRS

На высоком уровне шаблон «источники событий» — это всего лишь наблюдение, что события (т. е. изменения состояний) являются основными элементами любой системы. Так что, если они хранятся в неизменном виде, в том порядке, в котором они были созданы, то журнал событий обеспечивает всесторонний аудит того, что произошло в системе. Более того, мы в любой момент можем вывести текущее состояние системы, перематывая журнал и воспроизводя события по порядку.

CQRS — логическое продолжение этой идеи. В качестве простого примера, вы можете записывать события в Kafka (модель записи), считывать их обратно и отправлять в базу данных (модель чтения). В этом случае Kafka

асинхронно преобразует модель записи в модель чтения, разделяя их во времени, чтобы оптимизировать каждую из них независимо.

Шаблон «источники команд» — это один из видов шаблона «источники событий», но применяемый к событиям, которые попадают в сервис, а не создаются им. Звучит немного абстрактно, поэтому давайте посмотрим на пример на рис. 7.1, похожий на пример из предыдущей главы, где пользователь совершал покупку, а возникающий вследствие этого заказ проверялся и возвращался.

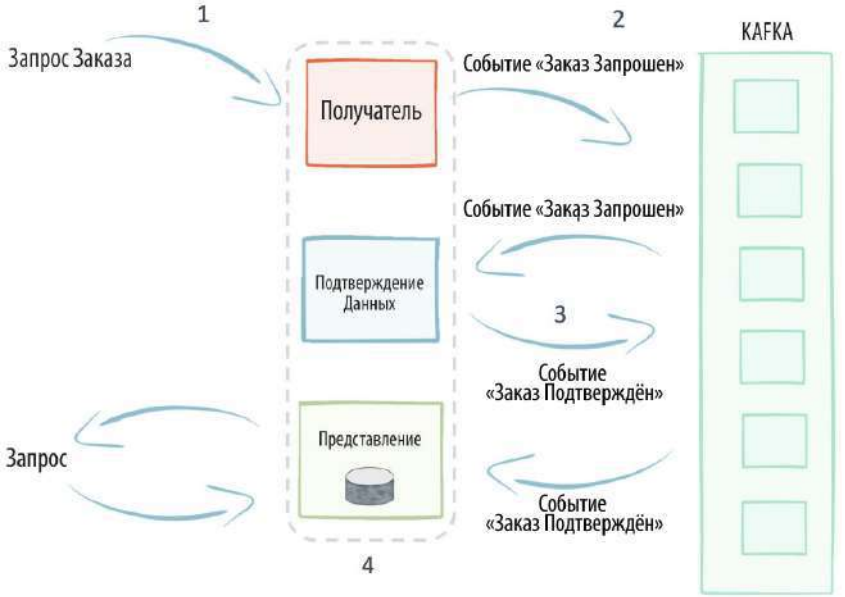


Рис. 7.1. Простая проверка заказа

При совершении покупки (1) шаблон «источники команд» требует, чтобы запрос заказа был немедленно сохранён в журнал как событие, прежде чем что-либо произойдёт (2). В этом случае, если что-то пойдёт не так, сервис можно вернуть к исходному состоянию, а затем воспроизвести его действия заново, например, для восстановления при повреждении.

Далее, заказ проверяется, и в журнале сохраняется новое событие, говорящее об изменении состояния (3). В отличие от обновления на месте (англ. update in place), как в моделях типа CRUD (англ. create, read, update, delete — создать, прочитать, обновить, удалить),[64] проверенный заказ не перезаписывает старый заказ, а представляет собой новое событие, которое записывается в журнал. В этом и есть суть шаблона «источники событий».

Наконец, чтобы запросить заказ, к результирующему потоку событий

добавляется база данных, которая происходит из представления источников событий и представляет собой текущее состояние заказов в системе (4). Так, (1) и (4) представляют собой стороны запросов и инструкций в CQRS.

Эти шаблоны обладают рядом преимуществ, которые мы рассмотрим в последующих разделах.

Контроль версий ваших данных

Сохранение событий в журнал напоминает систему контроля версий для данных. Рассмотрим ситуацию из рис. 7.2. Если появилась ошибка в программе, например, поле с отметкой времени было обработано с неверным часовым поясом, произойдет повреждение данных. Повреждённые данные попадут в базу данных. Они также попадут во взаимодействия сервисов друг с другом, тем самым распространяя ошибочные данные дальше и усложняя их исправление.

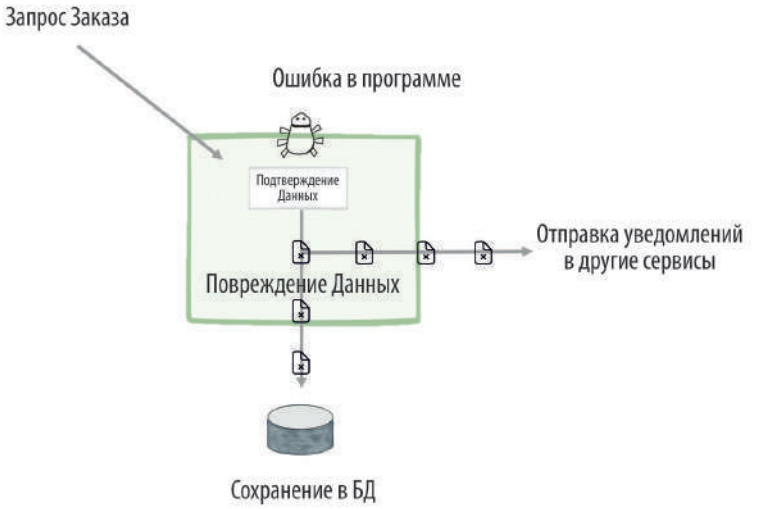


Рис. 7.2. Программная ошибка привела к повреждению данных как в базе данных сервиса, так и в данных, которые сервис отдавал наружу

Исправление такой ситуации нетривиально по нескольким причинам. Во-первых, оригинальный ввод данных в систему не был зафиксирован, и у вас есть только повреждённая версия заказа. Придётся исправлять данные вручную. Во-вторых, в отличие от системы контроля версий, которые могут отправить вас назад во времени, база данных изменяется неотвратимо, и предыдущие состояния системы потеряны навсегда. Таким образом, для сервиса не существует простого способа «отменить» нанесённый ущерб.

Чтобы исправить ситуацию, программисту придётся совершить ряд действий: применить исправление к программе, запустить скрипт базы данных для исправления ошибочных отметок времени и, наконец, придумать, как разослать сервисам правильные данные взамен испорченных. В лучшем случае это потребует написания нового кода, который выгружает данные из базы, исправляет их и заставляет сервисы послать запрос на передачу исправленных данных. Но, поскольку база данных записывает новые значения поверх старых, этого может оказаться недостаточно. (Если бы вместо исправления выпуска систему просто откатили назад после того, как она какое-то время поработала, то процесс миграции данных был бы, вероятно, ещё сложнее.)

Примечание: Воспроизводимый журнал превращает эфемерные сообщения в сообщения, которые хранятся в памяти в течение длительного времени.

Использование подхода «источники событий/команд», где и ввод, и изменения состояний записываются, может выглядеть примерно как на рис. 7.3.

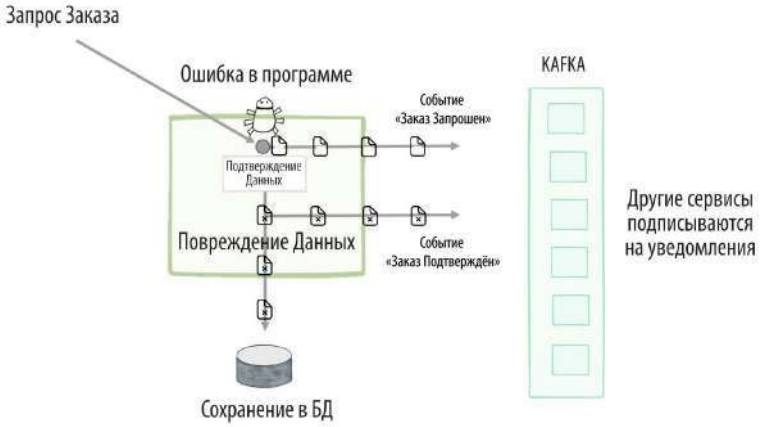


Рис. 7.3. Подключение Kafka и подхода «источники событий» к системе, показанной на рис. 7.2 гарантирует, что оригинальные события сохранились до того, как ошибочный код был исполнен

Поскольку Kafka хранит события так долго, как нам нужно (что мы обсуждали в разделе «Долговременное хранение данных» на стр. 28 в Главе 4), исправление повреждённых временных меток является тривиальной задачей. Сначала исправляется ошибка в коде, затем журнал отматывается до того момента, когда ошибка проявила себя, а затем система воспроизводится из потока запросов заказов. Правильные временные метки записываются в базу данных поверх испорченных значений, а новые

события отправляются в поток, исправляя предыдущие испорченные события. Способность хранить входные данные, способность перемотки и воспроизведения — всё это значительно облегчает восстановление системы после сбоев.

Подход «источники команд» позволяет записывать входящие данные. Это означает, что систему можно отмотать назад и воспроизвести в любой момент. Подход «источники событий» позволяет записывать изменения состояния. Это гарантирует, что мы точно знаем, что произошло во время выполнения действий в системе, и что мы всегда сможем восстановить текущее состояние (в данном случае — содержимое базы данных) из журнала изменений состояния (рис. 7.4).

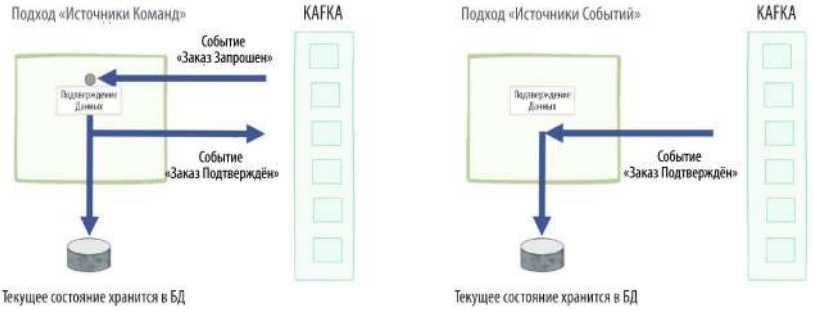


Рис. 7.4. Подход «источники команд» обеспечивает прямое повторение оригинальных команд, в то время как подход «источники событий» воссоздаёт текущее состояние системы по обработанным событиям из журнала

Способность хранить упорядоченный журнал изменений состояния полезен для отладки и отслеживания, отвечая на ретроспективные вопросы, вроде «Почему этот заказ был неожиданно отклонён?» или «Почему это вдруг баланс в минусе?». На такие вопросы сложно ответить, имея лишь изменяемое хранилище.

Примечание: Подход «источники событий», как и система контроля версий, гарантирует запись каждого изменения состояния системы. Как у нас говорится, «бухгалтеры не пользуются ластиками».

Стоит заметить, что есть и другие устоявшиеся шаблоны БД с некоторыми из этих свойств. Промежуточные таблицы[65] можно использовать для хранения непроверенных входных данных, триггеры[66] можно использовать в реляционных базах для создания таблиц аудита; битемпоральные базы данных[67] тоже предоставляют структуру данных для аудита. Все эти методы очень полезны, но ни в одном из них не заложено возможности «перемотки и повтора» без больших трудозатрат

со стороны программиста. Используя же подход «источники событий», в написании или тестировании нового кода практически нет необходимости. Один и тот же код используется как для нормальной работы, так и для восстановления.

Превращение событий в источник истины

Одним из побочных эффектов предыдущего примера является то, что источник событий, а не база данных, является источником истины. Из использования событий в качестве источника истины можно сделать интересные выводы.

Если продолжить пример с заказом, то выходит, что существует две версии заказа: одна в базе данных, а вторая — в уведомлении. Это приводит к двум разным проблемам. Первая заключается в том, что и уведомление, и запись в базе данных должны быть выполнены атомарно. Представьте, что поломка произошла на полпути между обновлением базы данных и отправкой уведомления (рис. 7.5). В лучшем случае это приведёт к созданию дубликата уведомления. В худшем — уведомление может вообще не создаться. Эта проблема может обостряться в сложных системах, как мы обсудим в Главе 12, рассматривая транзакции Kafka.

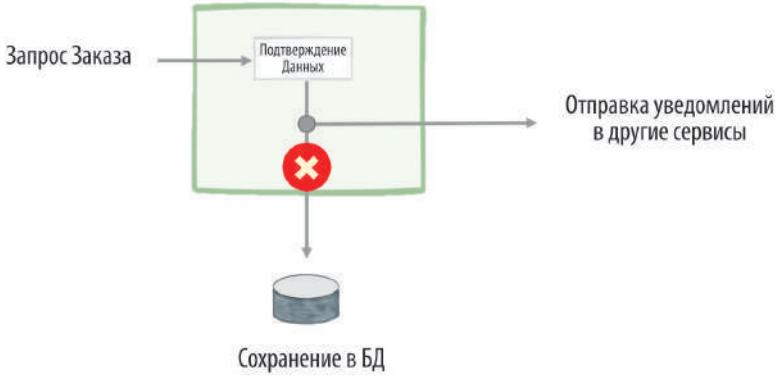


Рис. 7.5. Запрос заказа проверен, уведомление отправлено другим сервисам, но в самом сервисе произошёл отказ до того, как данные сохранились в базе

Вторая проблема состоит в том, что на практике, при добавлении кода и внедрении новых функций, данные в базе и в уведомлениях легко могут рассинхронизироваться. Даже если данные в базе данных верны, но они расходятся с уведомлениями, то страдает качество данных всей системы. (См. раздел «Проблема расхождения данных» на стр. 106 в Главе 10.)

Подход «источники событий» решает обе эти проблемы путём превращения

потока событий в основной источник истины (рис. 7.6). Когда необходимо запросить данные, модель чтения или представление (англ. view) потока событий выводятся непосредственно из потока.

Примечание: Подход «источники событий» гарантирует, что состояние, которое сервис передаёт дальше, и состояние, которое он сохраняет внутри, идентичны друг другу.

И это весьма логично. В традиционных системах база данных является источником истины. Это имеет смысл, когда вы смотрите на систему изнутри. Но если посмотреть на это с точки зрения других сервисов, то выходит, что им не важно, что хранится внутри. Важны те данные, которые видны снаружи. В этом смысле идея использовать события в качестве источника истины выглядит естественно. И это, в свою очередь, приводит нас к модели CQRS.

Командно-запросная изоляция

Как уже обсуждалось раньше, CQRS разделяет пути чтения и записи и соединяет их с асинхронным каналом (рис. 7-6). Эта идея не ограничивается архитектурой приложения, она подходит и для других применений. Базы данных, к примеру, используют идею журнала упреждающей записи (англ. write-ahead log).[68] Вставки и обновления немедленно записываются в журнал на диск в последовательном режиме. Это позволяет базе данных ответить пользователю, что данные получены и будут обработаны, не дожидаясь выполнения медленных операций, вроде обновления столбцов, индексов и прочего.[i] Смысл в том, что (а) если что-то пойдёт не так, внутреннее состояние базы данных можно будет восстановить из журнала, и (б) запись и чтение можно оптимизировать независимо друг от друга.

В приложениях CQRS используется по тем же причинам. Входящие данные журналируются в Kafka, что намного быстрее записи в базу данных. Разделение модели чтения и её асинхронное обновление означает, что трудозатратные операции по обновлению структуры данных на месте, например, индексация, могут быть объединены для более эффективного исполнения. Это означает, что CQRS покажет лучшую общую производительность в чтении и записи по сравнению с эквивалентными CRUD-системами.

i Некоторые базы данных, например, DRUID, делают это разделение явным. Другие базы данных блокируются до обновления индекса. См. Apache Druid (incubating) is a high performance real-time analytics database. *Apache Druid.* - URL: https://druid.apache.org/

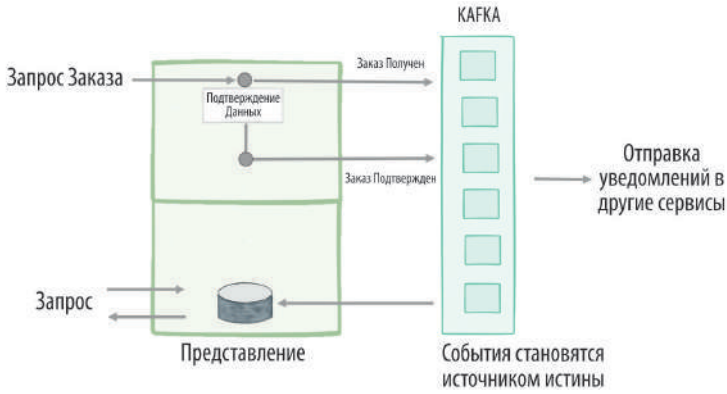


Рис. 7.6. Когда мы делаем поток событий источником истины, обновление базы данных и уведомление происходят из одного события, которое хранится в неизменном виде. Разделяя потоки чтения и записи, система реализует шаблон CQRS

Но, как известно, ничто не даётся даром. Подвох в том, что, поскольку модель чтения обновляется асинхронно, она слегка отстаёт от модели записи. Таким образом, если пользователь выполнит операцию записи, а затем немедленно выполнит операцию чтения, то может возникнуть ситуация, в которой изначально записанная сущность не успеет распространиться, и её будет невозможно тут же прочитать, как будто пользователь не может прочитать собственноручно записанные данные. Как мы увидим в примере в разделе «Нарушение принципа CQRS через блокировку чтения» на стр. 156 в Главе 15, существует несколько стратегий для решения этой проблемы.

Материализованные представления

Существует тесная связь между запросами CQRS и материализованным представлением в реляционных базах данных. Материализованное представление — это таблица, содержащая результат предопределённого запроса, где представление обновляется с каждым изменением соответствующей таблицы.

Материализованное представление используется в целях оптимизации производительности, чтобы производить вычисления заранее и предоставлять данные пользователю по необходимости, а не ждать запроса пользователя на выполнение вычислений. Например, если мы хотим отобразить количество активных пользователей на каждой странице сайта, нам потребуется просканировать базу данных и посчитать посещения, а это довольно ресурсоёмкая операция. Вместо этого можно

заранее вычислить результат нужного запроса, и, поскольку сам результат небольшой, его можно быстро извлечь при необходимости. Таким образом, это хороший кандидат на предварительное вычисление.

Можно создать точно такую же конструкцию, используя CQRS в Kafka. В Kafka записывается информация о командах (вместо прямого обновления таблицы в базе данных). Мы можем трансформировать поток событий таким образом, чтобы он подходил именно для нашего случая, с использованием Kafka Streams или KSQL, а затем материализовать его в виде предварительно рассчитанного запроса или материализованного представления. Поскольку Kafka работает по модели «издатель-подписчик», мы можем иметь множество представлений, которые будут вычисляться заранее для разных ситуаций (рис. 7.7). Но в отличие от материализованных представлений в реляционных базах данных, события, лежащие в их основе, будут отделены от самого представления. Это означает, что (а) они могут масштабироваться независимо и (б) процессу записи, который, например, записывает посещения пользователей, не нужно дожидаться вычисления отображения.

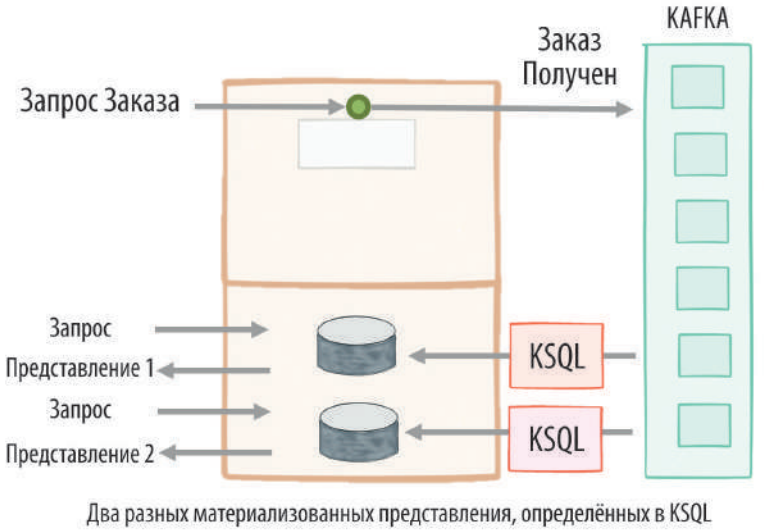


Рис. 7.7. CQRS позволяет использовать различные модели чтения, которые, как и материализованные представления, могут быть оптимизированы для конкретного случая, либо использовать другие технологии хранения

Идея хранения данных в журнале и создания производных представлений получает дальнейшее развитие в Главе 9 «Поток событий как общий источник истины».

Примечание:

Если поток событий является источником истины, то можно иметь множество представлений любых форм и размеров, какие вам только могут понадобиться. Каждое представление будет ориентировано на определённую задачу.

Полиглотные представления

Независимо от размера задачи, будь то полнотекстовый поиск, сбор аналитики, быстрый поиск ключей и значений или что-то ещё, всегда найдётся база данных именно для вашего случая. В то же время это означает, что к базам данных нет одного универсального подхода, по крайней мере пока нет.[69] Дополнительное преимущество использования CQRS состоит в том, что единая модель записи может отправлять данные во множество моделей чтения или материализованных представлений. Так что ваша модель чтения может находиться в любой базе данных, или даже в разных базах.

Воспроизводимый журнал облегчает запуск полиглотных представлений, настроенных на разные ситуации, из тех же данных (рис. 7.7). Типичным примером является быстрый поиск ключей и значений и сохранение их в очередь запросов с сайта, а затем использование поисковых систем вроде Elasticsearch или Solr для поддержки полнотекстового поиска.

Полный факт или изменения?

Вопрос, который возникает в событийно-ориентированных программах (и особенно в программах с источниками событий), состоит в следующем: должны ли события моделироваться как целые факты (например, заказ целиком, со всеми связанными с этим данными), либо как изменения, которые потом ещё нужно скомпоновать (сначала сообщение о целом заказе, затем сообщение только о том, что изменилось: «Сумма обновлена до 5 $», «Заказ отменён», и т. д.).

В качестве аналогии, представьте, что вы создаёте систему контроля версий вроде SVN или Git. Когда пользователь фиксирует файл в первый раз, система сохраняет его на диск целиком. Последующие фиксации, отражающие лишь изменения в файле, могут содержать только изменения — лишь добавленные, изменённые или удалённые строки. Затем, когда пользователь запрашивает определённую версию, система открывает файл нулевой версии и применяет все последующие изменения, чтобы вывести именно ту версию, которая нужна пользователю.
Альтернативным подходом является хранение файла целиком со всеми

необходимыми изменениями для каждой отдельной фиксации. Очевидно, что это потребует больше места для хранения, зато извлечение конкретной версии файла из истории будет происходить быстро и просто. Однако если пользователь захочет сравнить разные версии, системе придётся использовать функцию diff.

Эти же два подхода применимы к данным, хранимым в журнале. Возьмём более бизнес-ориентированный пример. Обычно заказ состоит из набора элементов (например, вы покупаете за раз несколько товаров). При внедрении системы обработки покупок вы могли задаться вопросом: стоит ли моделировать заказ как одно событие со всеми составными частями внутри него, или каждый элемент стоит выделить в отдельное событие, а заказ бы компоновался путём сканирования различных независимых частей? В предметно-ориентированном подходе (DDD)[70] заказ второго типа будет называться агрегатом (поскольку он является совокупностью частей), представляющим сущность заказ, который называют корнем агрегации.

Как и с другими проблемами в проектировании ПО, существует множество мнений о том, какой подход является наилучшим для каждого отдельного случая. Впрочем, есть несколько советов, которые помогут с выбором. Наиболее важный из них — журналирование целого факта после его прибытия. Поэтому, когда пользователь создаёт заказ, если в заказе присутствуют все составные части, мы обычно записываем его как единое целое.

Но что, если пользователь отменит одну из позиций в заказе? Простым решением было бы журналировать всё целиком ещё раз, как новое отменённое событие. Но что, если по какой-то причине заказ недоступен, и всё что у нас есть — это одна отменённая часть? Тогда возник бы соблазн посмотреть изначальный заказ внутри системы (например, через базу данных) и объединить его с отменой, чтобы создать новый «отменённый заказ», куда были бы внедрены все нужные элементы. Обычно это не самое лучшее решение, потому что (а) мы не записываем именно то, что получили, и (б) обращение к базе данных нивелирует преимущества производительности CQRS. Следуйте простому правилу: записывайте то, что получаете. Даже если это всего одна строка — записывайте её. Скомпоновать можно и в процессе чтения.

И наоборот, разделение событий на подсобытия по мере поступления — не лучшая практика по тем же причинам. Резюмируя, сформулируем основное правило: записывайте именно то, что вы получаете, без изменений.

Реализация источников событий и CQRS с помощью Kafka

Kafka поставляется с двумя разными API, которые помогают создавать и управлять представлениями в CQRS-стиле, произведёнными из событий, хранимых в Kafka. Kafka Connect API и связанная экосистема коннекторов[71] предоставляют готовый механизм отправки или получения данных из разных баз, источников и приёмников. Кроме того, Kafka Streams API поставляется с простой базой данных под названием «хранилище состояний», встроенной в API (см. «Оконные функции, соединения, таблицы и хранилище состояний» на стр. 150 в Главе 14.

В оставшейся части этого раздела мы рассмотрим полезные шаблоны для реализации источников событий и CQRS с помощью этих инструментов.

Создание представлений в Kafka Streams с помощью таблиц и хранилищ состояний

Streams API предоставляет один из простейших механизмов реализации источника событий и CQRS потому, что позволяет создавать представления, прямо внутри Kafka Streams API — без внешних баз данных.

В простейшем случае предполагается превращение потока событий Kafka в таблицу, которую можно опрашивать локально. Например, превратив поток событий «Покупатель» в таблицу «Покупатели», чтобы запрос `CustomerId` занимал всего лишь одну строку кода:

```
Ktable<CustomerId, Customer> customerTable = builder.table("customer-topic");
```

Эта строка делает сразу несколько вещей:

- Она подписывается на события в теме Покупатель.
- Она переходит к самому раннему смещению и загружает все события «Покупатель» в Kafka Streams API. Это означает загрузку данных из Kafka в ваш сервис. (Обычно уплотнённые темы используются для сокращения времени первой загрузки или загрузки при наихудшем сценарии.)
- Она отправляет данные в хранилище состояний (рис. 7.8), локальную дисковую хэш-таблицу, находящуюся в Kafka Streams API. В результате получается локальная таблица «Покупатели», которую можно опросить по ключу или условию.

Рис. 7.8. Хранилище состояний в Kafka Streams может использоваться для создания специфичных представлений прямо внутри сервиса

В Главе 15 мы рассмотрим набор примеров с кодом для создания разных типов представлений и хранилищ состояний, а также обсудим, как такой подход может масштабироваться.

Пишем данные в тему Kafka из базы данных с помощью Kafka Connect

Одним из способов внесения событий в Kafka является их запись в тему Kafka через таблицу базы данных. Строго говоря, такой подход не относится к шаблонам «источников событий» или CQRS, но не менее полезен.

На рис. 7.9 сервис заказов записывает заказы в базу данных. Записи конвертируются в поток событий с помощью Kafka Connect API. Это запускает нисходящую обработку, которая проверяет заказ. Когда событие «Заказ подтверждён» возвращается в сервис заказов, прежде чем вызов вернётся к пользователю, база данных обновляется с окончательным состоянием заказа.

Самый надёжный и эффективный способ достичь этого состоит в использовании техники отслеживания измененных данных (англ. change data capture, далее — CDC).[72] Большинство баз данных записывают изменения в журнал упреждающей записи, чтобы при возникновении ошибки состояние восстановилось оттуда. Многие базы так же предоставляют механизм отслеживания примененных изменяющих операций. Коннекторы, реализующие CDC, используют эти механизмы, переводя операции базы данных в события, доступные системам обмена сообщений, вроде Kafka. Поскольку метод CDC использует родную функциональность баз данных, он (а) эффективен, т. к. коннектор использует

не API запросов, а напрямую отслеживает файл журнала или активируется непосредственно на изменения, и (б) точен, т. к. задержка при вызове запросов через API базы данных — это потенциальная возможность пропуска событий, произошедших в этот момент.

Рис. 7.9. Пример записи в поток событий через базу данных

В экосистеме Kafka CDC пока доступен не для каждой из существующих баз данных, но список постоянно растёт. Популярные базы данных с поддержкой CDC в Kafka Connect это MySQL, Postgres, MongoDB и Cassandra. Существуют проприетарные коннекторы для Oracle, IBM, SQL Server и других систем. Полный список коннекторов есть на главной странице Connect.[73]

Пишем данные в тему Kafka из хранилища состояний с помощью Kafka Streams

Тот же шаблон внесения данных в тему Kafka через базу данных может быть реализован в Kafka Streams, где база данных заменяется на хранилище состояний Kafka Streams (рис. 7.10). Нам становятся доступны все преимущества записи через базу данных с CDC, а также несколько дополнительных:

- Используется локальная база данных с быстрым доступом.
- Поскольку хранилище состояний обёрнуто Kafka Streams, оно принимает участие в транзакциях, и события, опубликованные сервисом и записанные в хранилище состояний, атомарны.
- При использовании единственного API требуется меньше настроек (нет внешних баз данных, не нужно настраивать коннектор CDC).

Мы обсудим использование хранилища состояний для состояния приложения в разделе «Оконные функции, соединения, таблицы и хранилище состояний» на стр. 150 в Главе 14.

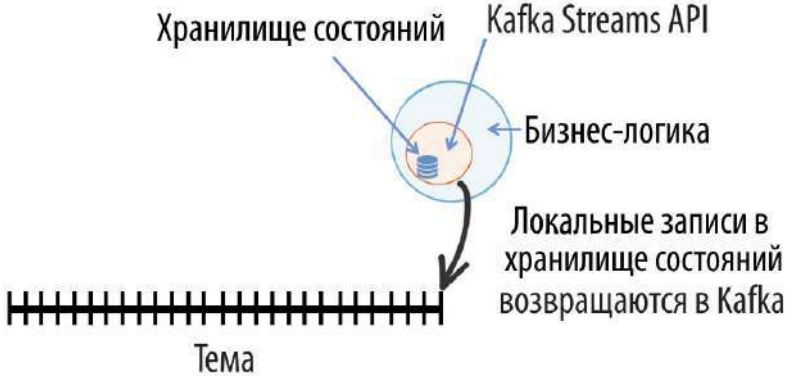


Рис. 7.10. Применение шаблона сквозной записи с Kafka Streams и хранилищем состояний

Интеграция со старыми системами с помощью CDC

В реальности многие проекты включают в себя устаревшие системы, и зачастую, несмотря на необходимость глобальных перемен в архитектуре, плавные изменения организовать несколько проще.

Проблема старых систем в том, что их очень сложно поменять. Но большинство бизнес-операций в старых системах по итогу всё равно сходится на базе данных. Поэтому не важно, насколько ужасен старый код, если база данных предоставляет способ внедрения[74] в бизнес-процесс, откуда события можно извлечь с помощью CDC. Как только у нас появляется поток событий, мы можем подключать новые событийно-ориентированные сервисы, которые позволят плавно уйти от наследия прошлого (рис. 7.11).

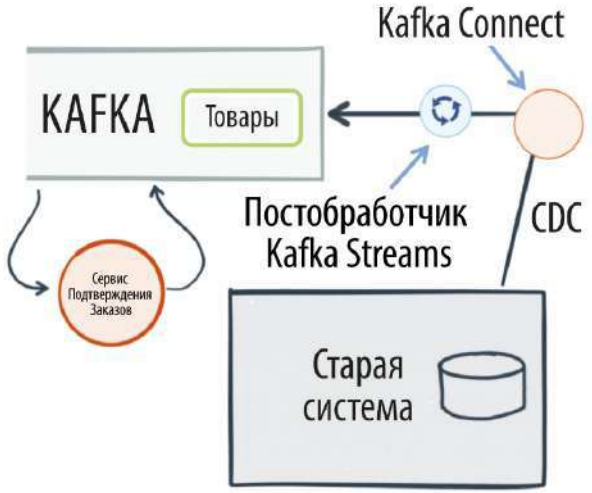


Рис. 7.11. Интегрируем старые данные с помощью API Kafka Connect

Итак, часть нашей устаревшей системы позволяет администраторам управлять каталогом продукции и обновлять его. Мы можем сохранить эту функциональность импортировав данные из базы старой системы в Kafka. Затем каталог продукции может быть использован повторно в сервисе проверки или в любом другом.

Проблема с подключением к устаревшим (или любым внешним) наборам данных в том, что данные зачастую плохо скомпонованы. В этом случае стоит рассмотреть добавление этапа постобработки. Для подобных операций отлично подходит трансформация отдельных сообщений[75] Kafka Connect (к примеру, для простых правок или дополнения), в то время как Kafka Streams API идеально подходит для разнообразных действий с потоками данных и для создания представлений для других сервисов.

Делаем запросы к оптимизированному для чтения представлению в БД

Ещё одним популярным шаблоном является использование Connect API для создания внутри базы данных представлений, оптимизированных для чтения. Такие представления можно создать быстро и без особых усилий в любых количествах и для любых баз данных, для которых доступны коннекторы Kafka Connect. Как мы уже обсуждали в предыдущем разделе, такие представления часто называют полиглотными, т. к. они дают доступ к огромному количеству технологий хранения данных.

В примере на рис. 7.12 мы используем Elasticsearch из-за его возможностей по полнотекстовому поиску, но в наши дни можно найти подходящую базу данных под любые типы ваших задач.[76] Еще одним распространённым шаблоном является предварительное вычисление материализованных представлений для данных с помощью Kafka Streams, KSQL или трансформации отдельных сообщений в Kafka Connect (см. «Материализованные представления» на стр. 68).

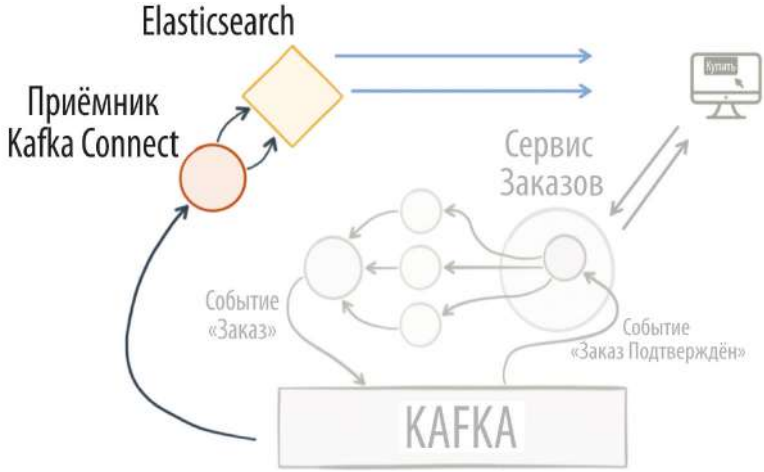


Рис. 7.12. Полнотекстовый поиск добавлен с помощью базы данных Elasticsearch, подключённой через Kafka Connect API

Образы в памяти и предзаполненные кэши

Наконец, следует упомянуть шаблон «образ в памяти» (англ. MemoryImage), рис. 7.13. Это всего лишь заумный термин от Мартина Фаулера[77] для обозначения кэширования всего набора данных в память, где он может быть опрошен, вместо использования внешней базы данных.

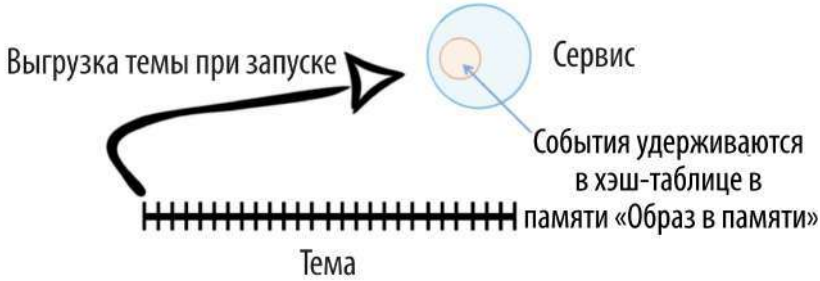


Рис. 7.13. Образ в памяти — это просто кэш целой темы, загруженный в сервис, что позволяет обращаться к ней локально

Образ в памяти обеспечивает простую и эффективную модель для набора данных, которая (а) помещается в память и (б) загружается за приемлемый отрезок времени. Для уменьшения времени загрузки для снимка журнала событий обычно используют уплотнённую тему (которая представляет последний набор событий без версионности). Шаблон «образ в памяти» можно организовать вручную или реализовать с помощью хранилищ состояний в памяти в Kafka Streams. Этот шаблон хорошо подходит для случаев высоконагруженных проектов, где нужно избегать перегрузки дисковой системы.

Представление источника событий

На протяжении оставшейся части книги мы будем использовать термин «представление источника событий» (англ. event-sourced view) для обозначения опрашиваемых ресурсов (базы данных, образы в памяти и т. д.), созданных в одном сервисе из данных, предоставленных другим сервисом.

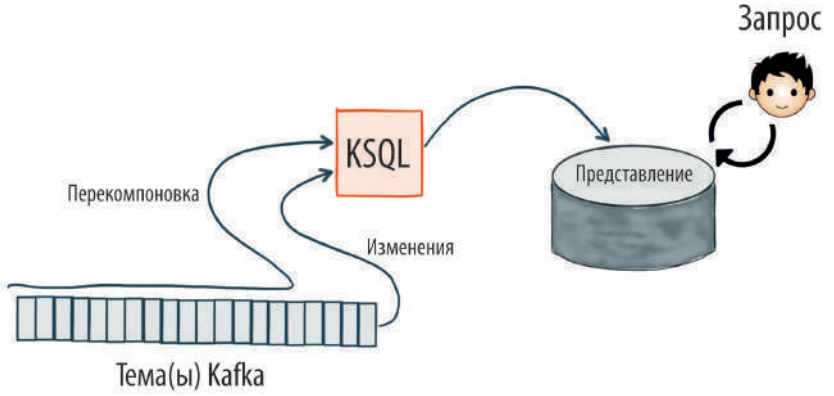


Рис. 7.14. Представление источника событий, реализованное с помощью KSQL и базы данных; при создании представления с помощью Kafka Streams и запрос, и представление будут существовать в одном и том же месте

Что отличает представление источника событий от привычной базы данных, кэша и подобных сущностей? В отличие от них оно может представлять данные в том виде, в котором это удобно для пользователей, сами данные добываются напрямую из журнала и могут быть перекомпонованы в любой момент.

К примеру, мы можем создать представление заказов, платежей и информации о покупателе, фильтруя всё, что не может быть доставлено в пределах выбранной страны. Такое представление будет представлением источника событий, если при изменении определения представления

(например, сменив страну доставки), оно автоматически пересоздастся из журнала событий.

В терминологии подхода «источников событий» представление источника событий эквивалентно проекции (англ. projection).

Заключение

В этой главе мы рассмотрели, как событие может быть чем-то большим, чем механизм для уведомлений или передачи состояний. Подход «источников событий» означает, что для записи состояния и для доступа к нему используется один и тот же механизм, при этом каждое изменение записывается беспремённо. Как мы отмечали в разделе «Контроль версий ваших данных» на стр. 63, это делает восстановление в случае сбоя или повреждения данных намного проще, чем при традиционных подходах к проектированию приложений.

CQRS идёт немного дальше, превращая «сырые» события в представление источника событий — конечную точку, доступную для опроса, которая компонуется напрямую из журнала (и может быть перекомпонована повторно). Важность CQRS заключается в том, что она масштабируется путём независимой друг от друга оптимизации моделей чтения и записи.

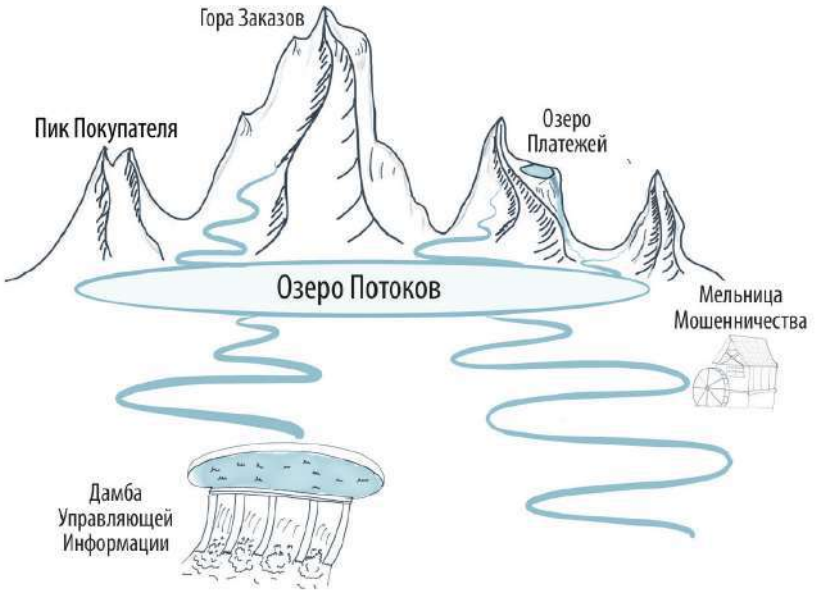
Затем мы рассмотрели различные шаблоны доставки событий в журнал, а также создания представлений с помощью интерфейсов Kafka Streams и Kafka Connect и одной из баз данных.

В конечном счёте, с точки зрения архитектора или рядового программиста, переход на использование подхода «источников событий» имеет большое влияние на архитектуру приложения. Источники событий и CQRS делают события по-настоящему важными участниками процесса. Это позволяет системам соотносить данные, живущие внутри сервиса, с данными, которые открыты для других участников. Позже мы увидим, как связать их вместе с помощью API Kafka Transactions. Мы также разовьём идеи из этой главы, применив их в межкомандном контексте, когда будем обсуждать подход «Поток событий как общий источник истины» в Главе 9.

ЧАСТЬ III

Переосмысление архитектуры в масштабе компании

Если присмотреться внимательнее, то можно увидеть, что все потоки данных и системы в
организации представляют собой единую распределённую базу данных.
— Джей Крепс



ГЛАВА 8

Совместное использование данных и сервисов в организации

Когда мы создаём программное обеспечение, наша основная задача, как правило, это решение какой-то реальной проблемы. Это может быть веб-страничка, функция для написания отчётов о продаже, аналитическая программа для выявления мошеннического поведения или любая другая функция или возможность, которая приносит ощутимую пользу людям. Это материальные цели, которые служат нашему бизнесу сегодня.

Но при создании ПО мы также думаем о будущем — но не путём разглядывания силуэтов в хрустальном шаре, предсказывая, где окажется компания в следующем году, а путём признания простого факта: что бы ни случилось, нашим программам придётся изменяться. Мы делаем это не задумываясь. Мы тщательно разделяем код на модули, чтобы он был понятен и мог использоваться повторно. Мы используем тесты, непрерывную интеграцию и, возможно, даже непрерывное развёртывание. Всё это требует усилий, и всё это слабо напоминает то, о чём пользователь программы может в явном виде попросить. Мы делаем это для того, чтобы наш код был долговечным. Но это отнюдь не значит, что мы должны держаться подальше от git push. Это значит, что мы вкладываем свои силы в код, который изменяется, развивается, перерабатывается и используется вновь. Старение в программном обеспечении — это функция не от времени, а от выбранного способа внесения изменений.

Но при проектировании системы мы редко задумываемся о её старении. Мы чаще задаёмся вопросами, вроде: «Будет ли система масштабируемой по мере роста количества пользователей?», «Будет ли время ответа достаточно быстрым, чтобы все были довольны?», «Поощряет ли система повторное использование?». Вы наверняка задумывались о том, как же всё-таки должна быть спроектирована долговечная система.

Если мы обратимся к истории, то увидим приложения для расчёта заработной платы на мейнфреймах, программы вроде Excel или Safari от гигантов индустрии или даже операционные системы вроде Windows или Linux. Но всё это сложные, независимые программы, которые были и остаются чрезвычайно важными для всего общества. Им тоже было сложно развиваться, особенно в плане организации больших технических усилий вокруг одной кодовой базы.[78] Если так сложно создать одну большую программу, то как же создать программу, которая будет работать на целую компанию? В данной главе мы рассмотрим именно этот вопрос: как спроектировать систему, которая будет взрослеть и расти вместе с компанией, помогая ей быть быстрой и полной сил?

Многие компании разумно начинают свой путь с одной системой, которая со временем становится монолитной, неповоротливой, и начинает напоминать большой комок грязи.[79] Самый распространённый ответ на возникновение подобной ситуации сегодня — это разбить монолитную систему на набор различных приложений и сервисов. В Главе 1 мы говорили о компаниях вроде Amazon, LinkedIn и Netflix, которые выбрали событийно-ориентированный подход к решению этой проблемы. Это не панацея. Более того, многие реализации микросервисных шаблонов подвержены заблуждению, что разделение системы на модули, связанные по сети, как-то само по себе улучшит её устойчивость. Разумеется, микросервисы не совсем про это. Но независимо от вашей интерпретации, само по себе разбиение монолита не повысит устойчивость. И тому есть несколько причин. Когда мы проектируем системы в масштабе компании, то фокусом системы становятся люди, а не программы.

По мере роста компании в ней формируются команды с разными сферами ответственности. Эти команды должны быть в состоянии работать без тесного взаимодействия друг с другом. Чем больше компания, тем больше требуется такой автономности. Это основа теорий управления вроде Slack.[80]

Но полная независимость также не будет работать. Разные команды и подразделения требуют определённого уровня взаимодействия или, по крайней мере, понимания общей цели. На самом деле, разделение социальных групп — это тактика, которую применяют и в политике, и на войне, как способ ослабления противника.[81] Хитрость в том, что нужно достичь организационного баланса между людьми, ответственностью и коммуникациями в компании. Это же применимо к программному обеспечению, потому что за пределами приложений главным, по-прежнему, остаётся человек.

Некоторые компании занимаются этим на организационном уровне, применяя подходы вроде «обратного приёма Конвэя».[82] Его идея состоит

в том, что если программное обеспечение и организация неразрывно связаны между собой (как утверждал Конвэй), то проще поменять организацию, и программное обеспечение последует за ним, чем делать наоборот. Но невзирая на принятый подход, при проектировании систем, где компоненты работают и развиваются независимо, у проблемы будет три составляющих — организация, ПО и данные, и все они неразрывно друг с другом связаны. Дело осложняется ещё и тем, что по-настоящему хорошие системы умеют управлять этими тремя факторами независимо, по мере их развития.

Это может звучать несколько абстрактно, но вам наверняка доводилось почувствовать действие этих факторов на себе. Скажем, вы работаете над проектом, но для завершения вам нужно дождаться окончания работы трёх других команд. Вы интуитивно понимаете, что в этом случае работа, планирование и выпуск займут больше времени. Если кто-то из другой команды спрашивает, может ли их приложение получить некоторые данные из вашей базы, вы точно знаете, что в будущем это приведёт к проблемам, ведь вы никогда не знаете, какую зависимость сломает очередное ваше обновление. Наконец, парочка запросов к REST-сервисам для наполнения пользовательского интерфейса может показаться тривиальной, но вы понимаете, что отказ на той стороне приведёт к звонку вам домой в три часа ночи. И чем сложнее становятся зависимости, тем сложнее будет разобраться, почему система не работает. Это всё примеры сложностей, с которыми мы сталкиваемся, когда организация, программное обеспечение и данные развиваются медленно.

В этом плане микросервисный подход чрезвычайно упрям. Он обрушивается на независимость организации, программ и данных. Микросервисами управляют разные команды, имеющие свои циклы развёртывания, которые не делятся кодом и базами данных.[83] Проблема в том, что заменить всё это на паутину RPC/REST вызовов не будет оптимальным решением. Это приводит нас к важному противоречию: мы хотим поощрять повторное использование для быстрой разработки программ, но тогда мы получаем больше зависимостей, что затрудняет любые изменения.

Примечание:

Повторное использование — это не всегда хорошо. Повторное использование помогает в быстрой разработке, но чем чаще используется компонент, тем больше это создаёт зависимостей, и тем сложнее такой компонент изменить.

Чтобы лучше понять это противоречие, нам нужно поставить под сомнение базовые принципы проектирования программ — принципы, которые

отлично работают при создании одного приложения, но дают сбой в ПО, разработка которого рассредоточена на несколько команд.

Инкапсуляция не всегда идёт на пользу

Нас, программистов, учили, что нужно всегда применять инкапсуляцию. Если вы делаете библиотеку для использования другими людьми, вы будете тщательно подбирать соглашение о функциях, которые вы хотите сделать доступными. Если вы делаете систему, основанную на сервисах, то, возможно, вы следовали похожей логике. Это работает, если вы чётко разделили ответственность между различными сервисами. К примеру, сервис единого входа (англ. single sign-on) имеет хорошо обозначенную роль, которая отличает его от ролей других сервисов (рис. 8.1). Это чёткое разделение означает, что даже в условиях резкой смены требований сервис единого входа вряд ли придётся менять. Он существует в строго ограниченном контексте.

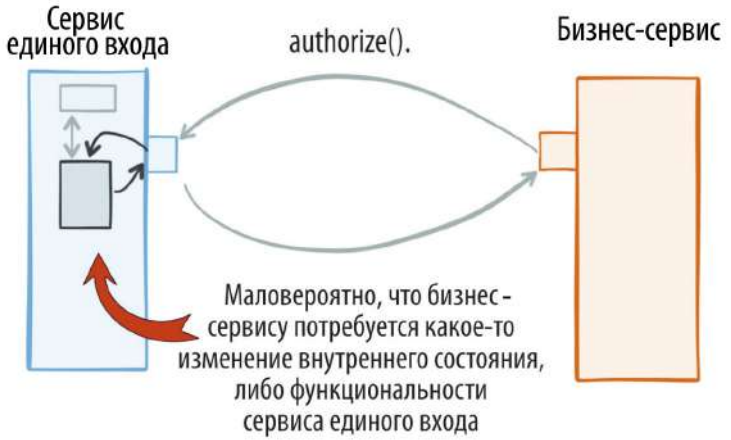


Рис. 8.1. Сервис единого входа — хороший пример инкапсуляции и повторного использования

Проблема в том, что в реальном мире бизнес-сервисы не могут быть так чётко разделены между собой. А это значит, что новые требования рано или поздно пересекутся с другими сервисами, и их придётся менять одновременно. Это можно измерить.[84] Если одной команде необходимо реализовать функцию, и это требует изменений в коде от другой команды, то понадобится внесение изменений в оба сервиса примерно в одно и то же время. В монолитной системе это довольно просто: вы делаете изменение и затем выпускаете обновление. Но в случае, когда нужно синхронизировать два независимых сервиса, всё становится намного сложнее. Координация

между командами и циклами обновлений подрывает гибкость.

И эта проблема не ограничивается сервисами. Общие библиотеки страдают от тех же проблем. Если вы работаете в торговле, вам может показаться разумным создать модель для покупателей, заказов и платежей, и их связей друг с другом. Вы можете включить простую логику для стандартных операций вроде возвратов и обменов. Многие поступали так на пике популярности объектно-ориентированного программирования, но последствия оказались довольно болезненными, потому что внезапно оказалось, что чувствительные части вашей системы тесно связаны с другими программами, и любые изменения даются очень тяжело. Поэтому микросервисы обычно не делят между собой одну предметную область. Но, разумеется, повторно использовать некоторые библиотеки не возбраняется. Например, библиотека журналов. Она, как и предыдущий пример с системой единого входа, вряд ли будет претерпевать значительные изменения.

Но, в целом, конечно, повторное использование библиотеки требует оговорки: код может использоваться где угодно. Скажем, вы использовали вышеупомянутую общую модель розничной торговли. Если пользоваться ей станет слишком сложно, всегда можно написать код самостоятельно! (Будет ли это хорошей затеей — тема отдельного разговора.) Но если подумать о других приложениях или сервисах, которые делят между собой данные, то подобного решения нет: без данных вы ничего не сможете сделать.

Это приводит к двум фундаментальным различиям между сервисами и общими библиотеками:

- Сервис управляется и используется кем-то другим.
- У сервиса обычно есть свои данные, тогда как библиотека (или база данных) ожидает данных от вас.

Суть проблемы, по-прежнему, в данных: большинство бизнес-сервисов значительно зависят от данных друг друга. Если у вас есть интернет-магазин, то поток заказов, каталог продукции или информация о покупателях неизбежно будут присутствовать во многих ваших сервисах. Каждый сервис нуждается в доступе к этим наборам данных для выполнения работы, и здесь не существует временного решения, чтобы без этих данных обойтись. Значит, вам нужен доступ к общим наборам данных, но при этом вы хотели бы связанность, по возможности, держать максимально слабой. Добиться этого не так-то просто.

Дихотомия данных

Инкапсуляция призывает нас скрывать данные, но системы данных имеют мало общего с инкапсуляцией. Даже наоборот: базы данных делают всё возможное, чтобы их данные были доступны другим (рис. 8.2). Они обладают мощным декларативным интерфейсом, который подстраивает форму данных под любые ваши требования. Это именно то, что нужно специалистам по обработке данных для исследований, но не совсем то, что нужно для управления межсервисными зависимостями.

Рис. 8.2. Сервисы инкапсулируют свои данные для снижения связанности и помощи в повторном использовании; базы данных множат свои данные для увеличения полезности для своих пользователей

Так мы оказываемся перед загадкой, дихотомией: базы данных нужны для того, чтобы данные были доступны и приносили пользу. А сервисы нужны для того, чтобы скрывать данные, избегая сильной связанности. Эти две силы являются основополагающими. Они определяют многое из того, что мы делаем, и борются за господство в системах, которые мы создаём.

Что происходит с системами по мере их развития?

По мере развития и роста систем мы можем наблюдать, что дихотомия данных проявляет себя двумя разными способами.

Проблема Главного сервиса

По мере роста сервисов данных они неизбежно увеличивают набор доступных извне функций, доходя до того, что начинают напоминать какую-то самодельную базу данных (рис. 8.3).

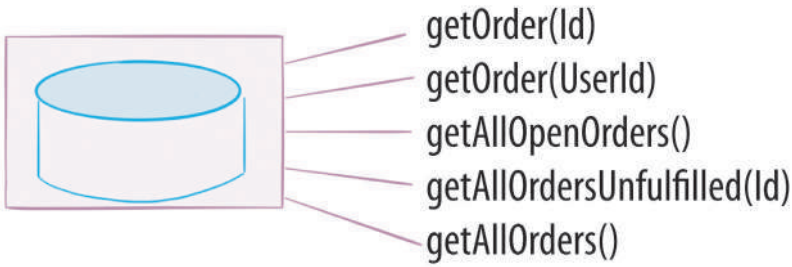


Рис. 8.3. Количество интерфейсов сервиса неизбежно растёт с течением времени

А создание чего-то похожего на безумную общую БД само по себе может привести к целому ряду проблем. Чем больше функций, данных и пользователей у таких сервисов, тем сильнее они связаны между собой, и тем сложнее их использовать и развивать.

Проблема REST-ETL

Вторая, более частая проблема состоит в том, что при встрече с подобным сервисом данных проще вытащить из него данные и работать с ними локально (рис. 8.4). На это есть множество причин, но можно выделить основные:

- Данные нужно комбинировать с другим набором данных.
- Данные должны быть ближе — например, по географическим причинам или для использования автономно (в мобильном устройстве).
- Возникновение проблем в сервисе данных может привести к отказу зависящих от него сервисов.
- Сервис данных не предоставляет необходимой функциональности, а изменения невозможны или слишком медленны.

Но, для извлечения данных из сервиса и поддержания их в актуальном состоянии потребуется некий механизм опроса. Конечно, это не так страшно, но и далеко от идеала.

Рис. 8.4. Данные массово перемещаются из сервиса в сервис

Более того, чем чаще происходят подобные перемещения данных в больших архитектурах, тем вероятнее проникновение небольших ошибок. Со временем ситуация лишь усугубится, и качество данных во всей экосистеме начнёт снижаться. Чем больше изменяемых копий, тем больше данные будут расходиться на большом временном отрезке.

В ретроспективе исправить несовпадающие наборы данных будет очень сложно. (Техники вроде управления основными данными (англ. master data management, далее — MDM),[85] по большому счёту, являются лишь временным решением.) На самом деле, самые сложные технологические проблемы, с которыми сталкивается бизнес, произрастают из расхождений в наборах данных, которые кочуют из приложения в приложение. Эта тема подробно обсуждается в Главе 10.

Циклический шаблон поведения возникает между (а) стремлением к централизации наборов данных для поддержания их в актуальном состоянии и (б) соблазном (или нуждой) извлекать данные и работать с ними в одиночку — и этому циклу неадекватности данных нет конца (рис. 8.5).

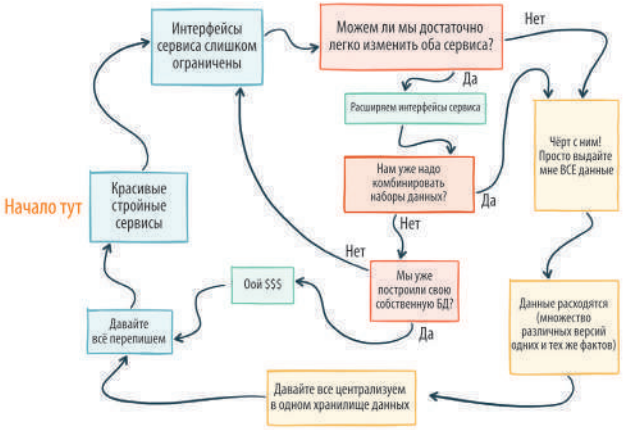


Рис. 8.5. Цикл неадекватности данных

Для решения этих проблем нужно рассматривать общие данные в несколько ином виде. Нужно обращаться с ними как с самой важной частью нашей архитектуры. Пэт Хэлланд в своих публикациях[86] подчёркивает, что с данными, которые делят между собой сервисы, и с внутренними данными нужно обращаться по-разному.

Внешние данные сложно изменить, потому что от них зависят многие другие программы. Но именно поэтому внешние данные являются самыми важными.

Вторая важная идея заключается в том, что команды, развивающие соответствующие сервисы, должны быть готовы к открытости: теперь сервису нужно не просто обслуживать приходящие запросы, но и быть частью широкой экосистемы. Это значительно отличается от традиционного подхода к созданию приложений, которые пишут для изолированной работы, а методы для взаимодействия и обмена данными добавляют потом.

С учётом этих особенностей становится понятно, что с внешними данными — данными, которые сервисы делят между собой, — необходимо обращаться осторожно и внимательно. Но чтобы получить свободу в произведении манипуляций над этими данными, нам потребуется перенести их внутрь и сделать своими.

Проблема в том, что ни один из существующих сегодня подходов, будь то интерфейсы сервисов, системы обмена сообщениями или общая база данных, не обеспечивают решения для подобного перехода (рис. 8.6) по следующим причинам:

- Интерфейсы сервисов формируют сильную связанность, что затрудняет обмен данными на любом масштабе, и оставляют без ответа вопрос: как соединить множество островков состояний вместе?
- Общие базы данных концентрируют все случаи использования в единственном месте, а это сдерживает развитие.
- Сообщения перемещают данные из сильно связанных мест (исходного сервиса) в слабо связанные места (сервис, использующий данные). Это значит, что наборы данных можно объединять, дополнять и манипулировать ими по мере необходимости. Перемещение данных в локальное хранилище увеличивает производительность, а также снижает связанность отправителя и получателя. К сожалению, системы обмена сообщениями не хранят никаких исторических данных, что означает сложность для запуска новых приложений. Кроме того,

со временем это может привести к снижению качества данных (что мы обсудим в разделе «Проблема расхождения данных» на стр. 106 в Главе 10).

Рис. 8.6. Компромисс между сервисными интерфейсами, системой обмена сообщениями и общей базой данных

Лучшим решением является использование воспроизводимого журнала, как в Kafka. Он работает как источник событий: наполовину система сообщений, наполовину база данных.

Примечание:

Сообщения делают из сильно связанных общих наборов данных (внешних данных) такие данные, которыми сервис может владеть и управлять (внутренние данные). Воспроизводимый журнал идёт ещё дальше, добавляя систему централизованного хранения исторических данных.

Не бойтесь развиваться

Когда вы начинаете новый проект, формируете новый отдел или открываете новую компанию, вам не обязательно делать всё абсолютно правильно с первого дня. Большинство проектов развивается постепенно. Они начинают как монолитные системы, затем к ним добавляются распределённые компоненты, появляются микросервисы и потоки событий. Важно понимать, что когда текущий подход будет вас ограничивать, вы сможете его изменить. Опытные архитекторы знают, где находится переломный момент. Если его пропустить, изменения могут стоить слишком дорого. Это тесно связано с понятием функции приспособленности (англ. fitness function) в эволюционных моделях.[87]

Заключение

Шаблоны вроде микросервисной архитектуры довольно категоричны в отношении независимости сервисов: сервисы зачастую используются разными командами со своими циклами развёртывания, своим кодом и

своими базами данных. Проблема в том, что замена всего этого на паутину RPC-вызовов не отвечает на важный вопрос: как сервисы получат доступ к этим островкам данных, помимо использования банальных запросов?

Дихотомия данных выделяет этот вопрос, подчёркивая противоречия между необходимостью сервисов оставаться слабо связанными и при этом контролировать, дополнять и комбинировать данные когда это требуется.

Это приводит к трём основным выводам:

1. с ростом архитектуры и ориентацией на данные перемещение данных из сервиса в сервис становится неотъемлемой частью развития системы;
2. внешние данные, т. е. данные, которые делят между собой сервисы, становятся самостоятельной единицей в процессе;
3. общая база данных не является приемлемым решением для хранения внешних данных, зато общий воспроизводимый журнал лучше уравновешивает эти проблемы, поскольку он может хранить наборы данных долговременно, и, кроме того, он облегчает событийно-ориентированное программирование, реагируя на текущие события.

Такой подход помогает синхронизировать данные между многими сервисами и через слабо связанный интерфейс даёт свободу вычленять, смешивать, дополнять и развивать данные локально.

ГЛАВА 9

Поток событий как общий источник истины

Как мы видели в Части II, события — это удобный инструмент для проектирования систем. Они предоставляют возможности отправки уведомлений, передачи состояний и снижения связанности. За последнюю пару десятилетий системы обмена сообщениями использовали эти свойства для перемещения событий из системы в систему, но лишь несколько лет назад системы сообщений стали использоваться в качестве слоя хранения, сохраняя наборы данных, которые они передают.[88] Это привело к появлению интересного архитектурного шаблона. Центральный массив данных компании хранится в виде централизованных потоков событий, которым присущи все атрибуты снижения связанности, встроенные в брокер сообщений. Но в отличие от традиционных брокеров, которые удаляют сообщения после прочтения, исторические данные сохраняются и доступны для любой команды, которой они понадобятся. Это приближает нас к идеям, развитым в Главе 7 (см. «Источники событий»), и концепции «данных снаружи» Пэта Хелланда.[89] ThoughtWorks называют этот шаблон потока событий «общим источником истины».[90]

База данных наизнанку

Фразу «выворачивание базы данных наизнанку» придумал Мартин Клеппманн.[91] По сути, идея заключается в том, что база данных обладает рядом компонентов — журналом транзакций, подсистемой запросов, индексами и кэшированием, и, вместо смешивания их в одном чёрном ящике (базе данных), мы разделяем их на отдельные части с использованием потоковой обработки, и они могут существовать в разных местах, работая вместе через журнал. Kafka играет роль журнала транзакций, потоковый процессор (вроде Kafka Streams) используется для

создания индексов или представлений, которые работают как непрерывно обновляемый кэш[92] внутри вашего приложения или рядом с ним (рис. 9.1).

Рис. 9.1. Потоковый обработчик и воспроизводимый журнал являются ключевыми элементами базы данных

В качестве примера можно представить этот шаблон в контексте простого приложения с GUI, которое позволяет просматривать заказ, платёж и информацию о покупателе в виде прокручивающейся сетки. Пользователь может прокручивать сетку вверх и вниз очень быстро, поэтому данные должны быть кэшированы локально. Но в потоковой модели, в отличие от периодического опроса базы данных и кэширования результата, нам понадобится определить представление для набора данных в прокручивающейся сетке, а потоковый процессор позаботится о его материализации. Поэтому вместо опроса базы данных и создания слоя кэширования поверх неё, мы в явном виде передаём данные туда, где они нужны, и обрабатываем их там (т. е. внутри GUI, прямо рядом с кодом).

Мы назвали «выворачивание БД наизнанку» шаблоном, хотя стоит считать это аналогией: это лишь ещё один способ объяснения потоковой обработки. И весьма мощный. Ещё одна причина, по которой эта аналогия находит отклик у людей, состоит в устоявшемся мнении, что держать бизнес-логику в базе данных — плохая идея. Но обратный процесс — передача данных в код — открывает огромные возможности для комбинирования данных и кода. Обработка потоков переворачивает традиционный подход к данным и коду, поощряя перенос данных в слой приложения для создания таблиц, представлений и индексов там, где они нам нужны.

Примечание:

«База данных наизнанку» — это аналогия обработки потоков, где те же самые компоненты — журнал транзакций, представления, индексы, кэши — не привязаны к одному месту, а доступны там, где они требуются.

Эта идея проявляется и в других областях. Сообщество Clojure говорит о деконструировании базы данных.[93] Это перекликается с источниками событий или полиглотами, как мы обсуждали в Главе 7. Но идея была изначально предложена Джеем Крепсом ещё в 2013 году, где он называл это «разделением», но в несколько другом контексте, и это оказалось очень важным:

> Прослеживается аналогия между ролью журнала для потока данных внутри распределённой базы данных и ролью, которую он играет для интеграции данных в больших организациях. Если присмотреться, то можно увидеть, что все потоки данных и системы в организации представляют собой одну распределённую базу данных. Можно рассматривать все индивидуальные запросно-ориентированные системы (Redis, SOLR, таблицы Hive и т. д.) только как конкретные индексы ваших данных. Можно рассматривать системы обработки потока вроде Storm или Samza как очень развитые механизмы триггеров и материализации. Я заметил, что приверженцы классических баз данных предпочитают такой угол обзора, потому что он понятно объясняет, что же люди делают со всеми этими разнообразными системами. Это всего лишь разные типы индексов![94]

Интересно, что Джей в своём описании проводит аналогию в контексте целой компании. Ключевым моментом является следующее: обработка потока отделяет ответственность за хранение данных от механизма, который данные запрашивает. Таким образом, может присутствовать один общий поток событий, скажем, для платежей, и множество специализированных представлений в разных частях компании (рис. 9.2). И это обеспечивает важную альтернативу традиционным механизмам интеграции данных. А конкретно:

- Журнал делает данные доступными централизованно в виде общего источника истины, но с наиболее простым подходом. Это обеспечивает слабую связанность приложений.
- Функция запросов не становится общей, она по-прежнему используется каждым сервисом отдельно, позволяя командам работать быстрее путём сохранения контроля над набором данных, который они используют.

Примечание:

Идея «базы данных наизнанку» важна потому, что воспроизводимый журнал вкупе с набором представлений является намного более гибкой структурой, чем единая общая база данных. Разные представления могут быть настроены на различные потребности, но все они восходят к одному источнику истины — журналу.

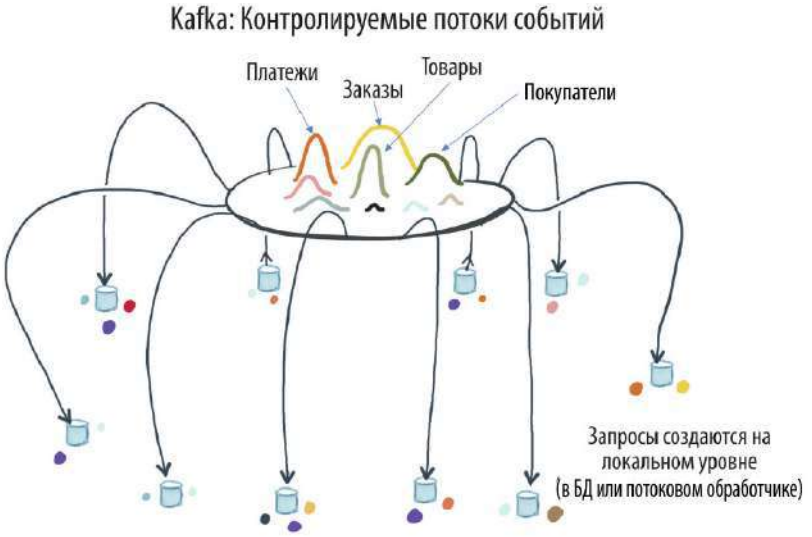


Рис. 9.2. Большое количество приложений и сервисов, каждый из которых использует своё представление, и все они являются производными от основного набора данных в Kafka; представления можно оптимизировать для разных ситуаций

Как мы увидим в Главе 10, это позволяет применить более глубокую оптимизацию, когда каждое представление настраивается на определённую ситуацию таким же образом, как материализованные представления в реляционных базах данных используются для оптимизации чтения пользовательских наборов данных. Конечно, в отличие от реляционных баз представление отделено от данных и может быть перестроено из журнала при необходимости внесения изменений. (См. «Материализованные представления» на стр. 68 в Главе 7.)

Заключение

В этой главе мы провели параллели между обработкой потоков и «базой данных наизнанку». По аналогии ответственность за источник данных (журнал) отделена от механизма запроса данных (Stream Processing API). Это позволяет создавать представления и внедрять их именно туда, где они нужны — в другие приложения, платформы или географические локации. Существует два основных мотива для передачи данных в код именно таким образом:

- в качестве оптимизации производительности путём хранения данных локально;

- для разделения данных в организации, но сохранения близости к единственному общему источнику истины.

Таким образом, на уровне организации шаблоны формируют подобие базы баз данных, где единственный источник данных о событиях используется во многих представлениях, и каждое представление может быть скорректировано в соответствии с требованиями команды.

ГЛАВА 10

Компактные данные

Суть компактных данных проста: вместо того, чтобы собирать и курировать огромные наборы данных, приложения тщательно выбирают небольшие, компактные данные, т. е. только те, которые им нужны в определённый момент времени, и которые отправляются из центрального источника событий в кэши или другие подконтрольные хранилища. Повторный вывод получившихся легковесных представлений данных совершается без каких-либо затруднений, так как такие представления легко поддерживаются операционными процессами.

Базам данных необязательно запоминать то, что уже хранится в системе обмена сообщениями

Интересным следствием использования потоков событий как источника истины (см. Главу 9) является то, что любые данные, полученные из журнала, не требуют обязательного сохранения. Потерянные данные всегда можно восстановить. Поэтому, если данные уже есть в слое обмена сообщениями, то базам данных их хранить необязательно (рис. 10.1). Следовательно, у вас есть возможность полностью восстановить базу данных или представление источника событий из журнала. Это может показаться странным. Зачем бы вам это понадобилось?

В контексте традиционной системы обмена сообщениями, процессов ETL (извлечение, преобразование, загрузка, англ. extract, transform, load) и тому подобного, слой обмена сообщениями недолговечен, и пользователи записывают все сообщения в базу данных сразу после того, как сообщения были прочитаны. В конце концов, они ведь могут никогда больше их не увидеть. Это приводит к нескольким проблемам. Архитектуры становятся довольно тяжёлыми, с огромным количеством приложений и сервисов,

сохраняющих копии большей части данных компании. На глобальном уровне по прошествии времени в этих копиях появляется всё больше расхождений, и перед нами встаёт проблема качества данных.

Проблемы качества данных не только широко распространены, но и зачастую имеют более серьезный характер, чем может показаться на первый взгляд.[95] На то есть несколько причин. Одна из них связана с нашим использованием баз данных как долгоживущих ресурсов, настройки и доработки в которых приводят со временем к непреднамеренно допущенным ошибкам. База данных — это, по сути, файл, который живёт столько, сколько живёт система, в которой он хранится. БД будет копироваться из среды в среду, и данные внутри неё претерпят немало операционных изменений. Поэтому не стоит удивляться ошибкам и неточностям.

В событийно-ориентированных системах с файлами такого не происходит. Представим, что обработчик потока данных создаёт представление, а затем выпускает обновление, меняющее вид этого представления. В этом случае первоначальное представление отбрасывается, обработчик откатывается к позиции 0 и создает новое представление из журнала.

Глядя на другие области нашей индустрии — DevOps и компанию, мы видим аналогичные шаблоны. Было время, когда системные администраторы индивидуально настраивали, отлаживали и видоизменяли компьютеры, работу которых они обеспечивали. В результате эти компьютеры приобретали незначительные, но все же отличия друг от друга, и при сбое было трудно понять причину неполадок.

Сегодня такие проблемы в значительной степени решены благодаря сервисным (англ. as-a-service) моделям обслуживания ПО, которые продвигают неизменяемость через подход «инфраструктура как код» (англ. infrastructure as code).[96] Этот подход имеет явные преимущества: развертывания становятся детерминированными, сборки — идентичными, а пересборки — простыми. И вот уже системные администраторы превращаются в счастливых людей, работающих с предсказуемыми инфраструктурами,[97] и уверенных в том, что их ПО будет функционировать всегда так же, как и во время тестирования.

Потоковая обработка предлагает аналогичный подход, но для данных. Представления источника событий содержатся в компактном объёме и выводятся из журнала детерминированным способом. Представление может быть кэшем, хранилищем состояний Kafka Streams или полноценной БД. Однако, чтобы это сработало, нам нужно разобраться с одной проблемой. Скорость загрузки данных может быть очень низкой. И в следующем разделе мы рассмотрим способы решения этого вопроса.

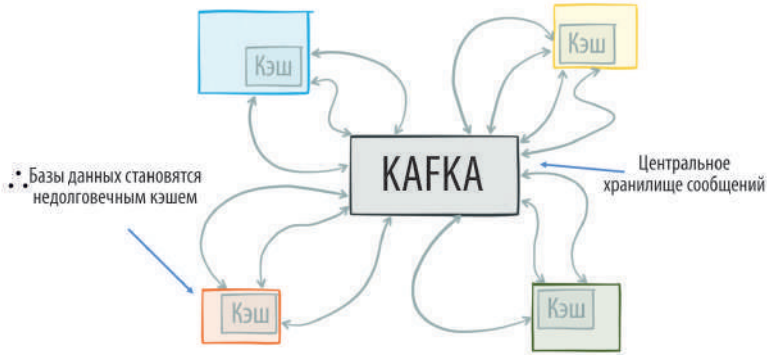


Рис. 10.1. Если система обмена сообщениями может хранить данные, то представления или базы данных могут этим не заниматься

Отбросьте всё лишнее, берите только необходимые данные

Если наборы данных хранятся в Kafka, тогда при внесении данных в ваш сервис вы можете выбрать только нужные части. Это позволяет не только уменьшить размер итогового представления, но оптимизировать его для чтения. Это аналогично тому, как материализованные представления используются в реляционных базах данных для оптимизации чтения. Однако, в отличие от реляционной базы данных, здесь запись и чтение не связаны. (См. «Материализованные представления» на стр. 68 в Главе 7.)

Сервис управления запасами, описываемый в Главе 15, является хорошим примером. Он считывает сообщения с большим количеством информации о запасах различных товаров, хранящихся на складе. Когда сервис читает каждое сообщение, он выбрасывает большую часть описаний объекта, оставляя только два поля: ID товара и количество товара на складе.

Сокращая представление, мы поддерживаем набор данных в компактном и слабосвязанном формате. Компактный формат набора данных позволяет хранить больше строк в располагаемой памяти или диске, а результатом, как правило, становится лучшая производительность. Связность также сокращается, поскольку в случае изменения схемы менее вероятно, что сервис будет хранить поля, затронутые произведёнными изменениями.

Примечание: Если система обмена сообщениями хранит все данные, создаваемые представления можно настроить на хранение лишь самых необходимых данных. Все остальные данные можно в любой момент восстановить.

Этот подход прост в реализации как с помощью Kafka Streams DSL или KSQL, так и с помощью трансформации отдельных сообщений Kafka Connect.[98]

Пересборка представлений источника событий

Очевидный недостаток компактных данных в том, что при необходимости большего количества данных вам придётся вернуться в журнал. Лучший способ сделать это — сбросить прежнее представление и создать новое с нуля. Если вы используете структуру данных в памяти, последнее произойдет по умолчанию. Если же вы используете Kafka Streams или базу данных, есть несколько факторов, требующих внимания.

Kafka Streams

При создании представления источника событий с помощью Kafka Streams полное воссоздание представления — это то, с чем придется смириться. Представления данных — это либо таблицы, являющиеся точной материализацией темы Kafka, либо хранилища состояний, заполненные результатами некоторых декларативных преобразований, написанные кодом JVM или KSQL. Оба типа автоматически восстанавливаются, если диск внутри сервиса потерян (или перемещён) или если задействована опция сброса потоков (англ. streams reset).[99] Более подробно о том, как Kafka Streams управляет контекстозависимостью, мы говорили в разделе «Практические вопросы контекстозависимости» на стр. 58 в Главе 6.

Базы данных и кэши

В современном мире существует множество различных типов баз данных, обладающих разными преимуществами и недостатками. В то время, как воссоздание пятидесятитерабайтной базы данных Oracle с нуля едва ли можно назвать целесообразным, восстановление представления источника событий, используемого в бизнес сервисах, является, зачастую, вполне рабочей схемой при правильном выборе технологии.

Чтобы минимизировать время восстановления при худшем сценарии, необходимо выбирать БД или кэш, оптимизированные для записи. Есть много вариантов, но разумный выбор включает в себя:

- кэш/БД в памяти, такие как Redis, MemSQL или Hazelcast;
- оптимизированные по использованию памяти базы данных, например Couchbase, или те, что позволят вам отключить ведение

журнала, наподобие MongoDB;
- оптимизированные для записи на диск, основанные на журналах базы данных типа Cassandra или RocksDB.

Решение затруднений при перемещении данных

Восстановление представления источника событий по-прежнему может быть довольно утомительной процедурой (и занимать минуты и даже часы![i]). Из-за этого, при выпуске нового ПО, разработчики обычно пересобирают представления заранее (или параллельно), переходя от старого представления к новому в тот момент, когда представление полностью сформируется. Это по сути тот же подход, который используют приложения контекстно-зависимой потоковой обработки Kafka Streams.

Такой шаблон хорошо работает для простых сервисов, которым требуются малые и средние наборы данных. Работа с большими наборами данных подразумевает более длительное время загрузки. Если вам нужно восстановить терабайтные объемы данных с одного сервера из базы данных, расположенной на диске и имеющей большое количество индексов, загрузка может занять непростительно много времени. Но во многих общих случаях ускорить процесс загрузки данных помогут решения, завязанные на памяти или предлагающие горизонтальное масштабирование.

Автоматизация и миграция схем

Системные администраторы, как правило, теряют доверие к инструментам, которые используются нерегулярно. Подобное недоверие еще сильнее к скриптам, которые оперируют данными (всем печально известны сценарии отката БД). Поэтому при переходе от среды к среде в процессе разработки, зачастую лучше воссоздавать представления данных напрямую из журнала, а не копировать файлы БД, как это принято при работе с традиционными базами данных (рис. 10.2).

Рассмотрим пример изменения схемы. Если вы до этого использовали традиционные подходы обмена сообщениями для переноса данных из одной системы в другую, вы скорее всего сталкивались с ситуацией,

i Например, RocksDB (используемая Kafka Streams) загружает ежеминутно примерно 10 миллионов объектов по 500 Кбайт (примерная скорость GbE) (См. Marotto F. Performance Benchmarks. - URL: https://github.com/facebook/rocksdb/wiki/performance-benchmarks#test-1-bulk-load-of-keys-in-random-order). Badger может загрузить примерно 4 миллиона объектов по 1 Кбайту в минуту на SSD (См. Rai Jain M. Introducing Badger: A fast key-value store written purely in Go. *Dgraph Logo*. May 14, 2017. - URL: https://blog.dgraph.io/post/badger/). А скорость загрузки Postgres - примерно 1 миллион строк в минуту (См. Mc Elhinney C. Mobile Mapping Spatial Database Framework. *LinkedIn*. June 17, 2011. - URL: https://www.slideshare.net/conormc/mobile-mapping-spatial-database-framework).

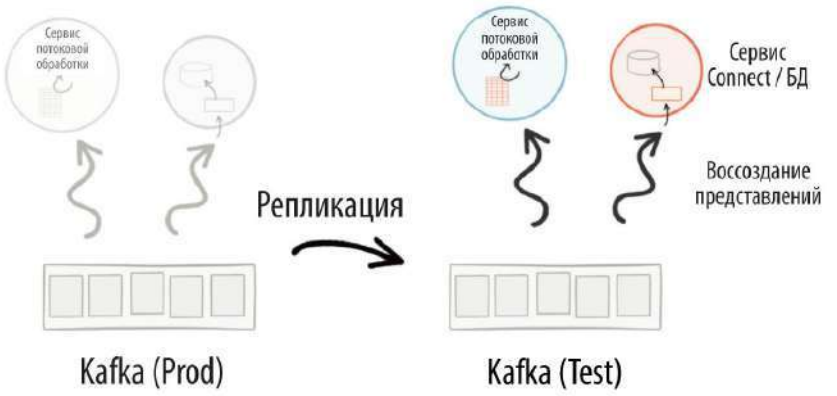


Рис. 10.2. Данные реплицируются в тестовое окружение, где представление данных восстанавливается из источника

когда совместимость старых и новых версий схемы обмена сообщениями нарушалась после перехода в новую среду. Например, вы импортируете информацию о клиентах из системы обмена сообщениями в БД в то время, когда схемы обмена сообщениями клиента претерпевают критическое изменение. В этом случае, как правило, вы сначала создадите скрипт БД для переноса данных, а затем подпишетесь на новую тему сообщений.

При использовании Kafka для сохранения целостности набора данных[ii] вы можете просто удалить старое представление и собрать новое, а не выполнять полномасштабную миграцию схемы.

Проблема расхождения данных

На практике все крупные растущие компании сталкиваются с проблемами качества данных. Эта проблема удивляет своей глубиной и сложностью, а вытекающие из нее вопросы требуют усердной, кропотливой и дорогостоящей проработки.

Причины этих вопросов кроются в разных местах. Рассмотрим типичное бизнес-приложение, которое получает данные из ряда различных систем и хранит их в БД. Модель обработки данных может быть текстовой (например, JSON), может опираться на внутренний домен (например, объектная модель), может служить описанию структуры БД (например, язык описания данных, англ. data definition language), и, наконец, отвечать за внешние коммуникации. Код должен быть написан для каждого из этих

ii При условии, что наборы данных были перенесены с использованием техники параллельного использования двух схем, описанной в разделе «Управление изменениями в схеме данных и нарушение обратной совместимости» на стр. 136 в Главе 13.

преобразований и должен видоизменяться согласно изменению различных схем. Каждый такой слой и требуемое индивидуальное понимание каждого такого слоя порождают риски неправильного толкования.

Семантические вопросы, часто возникающие при слиянии команд, отделов или учреждений, решить еще сложнее. Рассмотрим две компании в процессе слияния. Будет создан ряд эквивалентных наборов данных, смоделированных по-разному каждой компанией. Можно подумать, что это простая проблема трансформации, но, как правило, здесь кроются гораздо более глубокие смысловые противоречия: является ли поставщик заказчиком, а подрядчик — наёмным работником? В результате открывается широкое поле для ошибочного понимания.

Итак, по мере перемещения данных из приложения в приложение, из сервиса в сервис, процесс становится похожим на игру «Сломанный телефон». Первоначально корректные данные по истечению времени и после перехода от одного игрока к другому модифицируются, неправильно истолковываются и в финале предстают совершенно в другом виде.

Некоторые из вытекающих проблем не могут не беспокоить: несогласие Департамента Рисков с Департаментом Развития по поводу стратегии банка не дают компании развиваться; а недельная задержка с ответом на вопрос клиента из-за медленного делопроизводства не обещает продавцу каких-либо выгод. Все крупные компании могут поделиться подобными историями в той или иной форме. И хотя это не та проблема, с которой сталкиваются разработчики малых веб-приложений, эта проблема беспокоит создателей большинства крупных и более зрелых архитектур.

Есть испытанные и проверенные методы решения таких проблем. Некоторые компании создают системы сверки, которые могут превратиться в маленькие самостоятельные подразделения. Существует целая индустрия, посвящённая борьбе с этой проблемой, известная как MDM.[100] Также не стоит забывать о целом наборе инструментов для сведения данных к общему, согласованному основанию.

Платформы потоковой обработки данных помогают решить эти проблемы другим способом. Во-первых, они образуют своего рода центральную нервную систему, которая соединяет все приложения в один общий источник истины, сокращая риск «сломанного телефона». Во-вторых, учитывая, что данные сохраняются в журнале без изменений, отследить время совершения ошибки можно быстрее, а применить исправления с исходными данными на руках — легче. Поэтому, несмотря на то, что мы всегда будем совершать ошибки и допускать неверные толкования, такие приёмы, как превращение событий в источник истины или источник команд и использование компактных данных позволяют отдельным группам специалистов достичь оперативной зрелости, необходимой как

для исправления уже случившихся ошибок, так и для их предотвращения в будущем.

Заключение

Когда дело доходит до данных, ни о каких компромиссах не может быть и речи. В конце концов, они представляют суть нашего бизнеса. Практики компактной обработки и хранения данных позволяют нам всегда иметь под рукой общие факты и создавать представления, специально оптимизированные для решения конкретной проблемы. Факты не перегружают систему и легко воссоздаются, если следовать подходам «источников событий» и «источников команд», которые мы обсуждали в Главе 7. И хотя эти общие факты могут претерпевать изменения с течением времени, а также после применения и исправления в разных контекстах, они всегда будут связаны с единым источником истины.

ЧАСТЬ IV

Согласованность, параллелизм, эволюция парадигм

Без согласованности не может быть доверия

— Линкольн Чейфи

ГЛАВА 11

Согласованность и параллелизм в событийно-ориентированных системах

В информационных технологиях термин «согласованность» используется повсеместно, при этом его значение может меняться в зависимости от контекста. Согласованность в теореме CAP (англ. consistency, availability, and partition tolerance)[101] отличается от согласованности в рамках требований ACID (англ. atomicity, consistency, isolation, durability)[102] к транзакционным системам. Кроме того, существует целый спектр уровней согласованности, в частности, строгая согласованность[103] и итоговая согласованность (англ. eventual consistency).[104] Отсутствие у этого понятия согласованной терминологии может показаться несколько ироничным, но на самом деле отлично отражает его комплексный характер, обсуждение которого выходит далеко за рамки этой книги.[i]

Несмотря на эти многочисленные нюансы, у большинства людей сформировалось интуитивное представление о том, что же такое согласованность. Зачастую оно основано на опыте написания однопоточных программ[ii] или использования базы данных. Во втором случае представление о согласованности обычно исходит из общего

i Подробнее см. в энциклопедической книге Мартина Клеппманна Designing Data-Intensive Applications. O'Reilly. 2017.

ii Различные модели согласованности на самом деле отражают многочисленные варианты оптимизации последовательного исполнения команд на единственной копии данных. На практике такая оптимизация необходима, и большинство пользователей предпочтут отказаться от строгой согласованности в обмен на более высокую производительность (или доступность) системы. Поэтому были разработаны различные способы ослабить обычную «последовательную» согласованность. В результате оптимизации ряда параметров, которых особенно много в распределённых системах, появилось множество моделей согласованности. Когда мы будем обсуждать «Масштабирование параллельных операций в потоковых системах» (стр. 156, Глава 15), то увидим, как потоковые системы гарантируют строгую согласованность, разбивая соответствующие данные на разные потоки, а затем группируя эти операции в одну транзакцию. Это обеспечивает последовательное выполнение конкретной операции на единственной копии данных (т. е. реализуется модель строгой согласованности).

понимания транзакционных гарантий, которые предоставляет база данных. Когда мы добавляем запись в базу данных, то подразумеваем, что запись будет сохранена. Когда мы считываем запись, то подразумеваем, что получим самую последнюю версию значения. Если в рамках одной транзакции выполняется несколько операций, то все они становятся видимыми одновременно. Не нужно беспокоиться о том, что в то же время делают другие пользователи. Эту концепцию можно назвать «интуитивная согласованность» (технически наиболее близка к знаменитым требованиям ACID).

Общий подход к построению бизнес-систем состоит в том, чтобы непосредственно применить это интуитивное понимание согласованности. При разработке традиционных трёхзвенных приложений (клиент, сервер, база данных), обычно именно так и делается. Одновременные изменения обрабатываются базой данных изолированно от остальных пользователей. В любой заданный момент времени все пользователи видят одни и те же данные. Но группы сервисов обычно не гарантируют такой строгой согласованности. Микросервисы могут обращаться друг к другу синхронно, но в момент передачи данных от сервиса к сервису пользователи могут получать разные значения одних и тех же параметров. Этого не случится при наличии в архитектуре единой базы данных, которая координирует работу всех сервисов и реализует единую модель глобальной согласованности.

Но в мире, где приложения распределены (по географическим регионам, устройствам и т. д.), единая модель глобальной согласованности не всегда хороший выбор. Например, если данные создаются на мобильном устройстве, то они могут быть согласованы с данными на сервере только тогда, когда это устройство будет подключено к серверу. Если подключения нет, данные по определению будут не согласованы (по крайней мере, в этот конкретный момент), а синхронизация произойдет позже. Тогда и наступит итоговая согласованность. Опять же, важно проектировать системы так, чтобы учитывать временные периоды несогласованности. Так, в случае мобильного приложения, желательно, чтобы оно могло работать оффлайн, а затем, при возобновлении подключения к сети — выполнять синхронизацию с сервером, восстанавливая тем самым согласованность данных. Конечно, полезность оффлайн-режима зависит от конкретной задачи, которую необходимо выполнить. Возьмём, например, мобильное приложение интернет-магазина. В отсутствии подключения к сети приложение позволит вам выбрать конкретные продукты, но оно не сможет определить, доступны ли эти товары для заказа. Кроме того, оно не даст завершить покупку, пока вы снова не вернётесь в сеть. Возможность работы в оффлайн-режиме — один из примеров, когда строгая глобальная согласованность нежелательна.

Бизнес-системы совсем не обязательно проектировать для работы без сети схожим образом, но строгая глобальная согласованность и распределённые транзакции ведут к сложной и дорогой масштабируемости систем, они плохо работают в условиях географической распределённости и обычно достаточно неповоротливы.[105] Например, опыт с XA (англ. eXtended architecture, двухфазный протокол фиксации транзакции)[106] для работы с распределёнными транзакциями, привел к тому, что большинство реализаций транзакционных систем спроектированы с использованием очень сложных методов координации.[107]

Но с другой стороны, разработчики бизнес-систем, как правило, хотят соблюдения именно строгой согласованности, чтобы уменьшить вероятность ошибок. Таким образом, для многих безопасность всё-таки важнее скорости.[108] А кому-то хочется и того, и другого. Именно там, посередине и находятся событийно-ориентированные системы, в которых зачастую в той или иной форме используется итоговая согласованность.

Итоговая согласованность

В предыдущей части мы говорили об интуитивном понимании согласованности: операции выполняются последовательно на единственной копии данных. Систему с таким свойством построить достаточно легко. Сервисы обращаются друг к другу через RPC, точно так же, как работает вызов методов в однопоточной программе — это всё последовательные действия. Данные передаются по ссылке, через идентификатор. Сервис по идентификатору берет необходимые данные в БД. Если ему необходимо их изменить, он меняет их в базе. Пример такой системы представлен на рис. 11.1.

Этот подход интуитивно понятен, так как всё происходит последовательно. Однако по мере увеличения количества и размеров сервисов управлять такой системой становится всё сложнее.

Событийно-ориентированные системы обычно имеют другую архитектуру. Они используют асинхронные потоки, намеренно устраняя потребность следить за глобальным состоянием, а также избегая синхронного исполнения команд. (Мы обсуждали проблемы, связанные с общим глобальным состоянием в Главе 8 «Что происходит с системами по мере их развития?» на стр. 88.) Принято считать, что такие системы реализуют «итоговую согласованность».

Рис. 11.1. Набор сервисов, которые уведомляют друг друга и обмениваются данными через БД

Есть два последствия такой согласованности в данном контексте:

Своевременность
: Когда два сервиса обрабатывают один и тот же поток событий, они делают это с разной скоростью, то есть один сервис может отставать от другого. Если по какой-либо причине одновременно опрашиваются оба сервиса, это может привести к несогласованности.

Коллизии
: Если разные сервисы вносят изменения в одну и ту же сущность в одном и том же потоке событий, и эти данные позже объединяются, например, в базе данных, некоторые изменения могут быть потеряны.

Рассмотрим эти проблемы подробнее. В качестве примера снова возьмем интернет-магазин. На рис. 11.2 сервис заказов принимает заказ (1). Затем заказ передаётся на сервис подтверждения (2). Добавляется налог с продаж (3). Отправляется письмо на электронную почту пользователя (4). Обновлённый заказ возвращается в сервис заказов (5). Оттуда информация о заказе может быть запрошена через модуль просмотра заказов (это реализация CQRS, которую мы обсуждали в Главе 7). После того, как письмо с подтверждением было отправлено (6), пользователь может перейти по ссылке и просмотреть заказ (7).

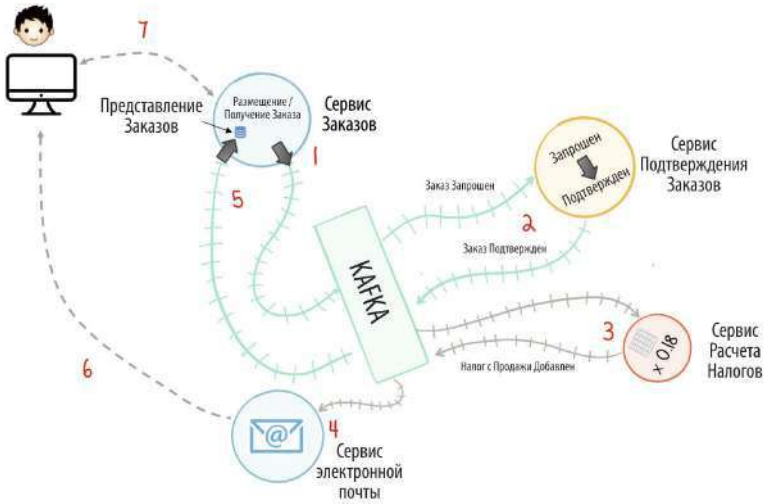


Рис. 11.2. Событийно-ориентированная система, объединённая через журнал событий

Своевременность

Рассмотрим сервис электронной почты (4) и сервис представления заказа (5). Оба подписаны на один и тот же поток событий («Заказ подтверждён») и обрабатывают их одновременно. Одновременное исполнение означает, что один из сервисов может немного отставать от другого. Безусловно, если бы мы прекратили добавлять новые записи в систему, сервис представления заказа и сервис электронной почты в конечном счете достигли бы согласованности. Однако обычно они находятся на несколько разных позициях в потоке событий. Им не хватает синхронизации между собой. Это может вызвать проблемы для пользователя, так как между сервисом электронной почты и сервисом представления заказов есть косвенная связь. Если пользователь щёлкнет по ссылке в электронном письме, но при этом модуль представления заказов еще не обновит данные, то ссылка либо не будет работать, либо вернёт ошибку (7).

Таким образом, рассинхронизация (в данном случае — отставание) в изолированных сервисах может вызывать проблемы, если они взаимодействуют каким-то образом между собой. Но, несмотря на это, для более крупных систем изоляция сервисов предпочтительнее, т. к. изоляция позволяет им работать одновременно и не мешать друг другу, а проблемы, связанные с отставанием, чаще всего можно обойти (этот вопрос тесно связан с обсуждением CQRS в Главе 7).

Но что, если такое поведение недопустимо, как в примере с электронной

почтой? В таком случае всегда можно вернуться к последовательному исполнению. Можно заблокировать вызов сервиса заказов до тех пор, пока модуль представления заказов не обновит данные (этот подход рассматривается на реальном примере в Главе 15). Другой вариант: сервис заказов создаст событие «Представление обновлено», которое будет запускать сервис электронной почты после того, как представление обновится. Оба способа используют последовательное исполнение там, где это необходимо.

Коллизии и объединение сообщений

Коллизии возникают, когда два сервиса одновременно обновляют одну и ту же сущность. Если система работает последовательно, то это исключено. Однако при параллельном исполнении команд такое возможно.

Рассмотрим сервис подтверждения заказов и сервис расчета налога на рис. 11.2. Если нужно, чтобы они запускались последовательно, можно написать программу таким образом, чтобы сначала работал сервис расчета налогов (в ответ на событие «Заказ запрошен»), а затем сервис подтверждения заказов (в ответ на событие «Налог с продажи добавлен»). Это преобразует обработку каждого заказа к линейному виду и означает, что конечное событие будет содержать полную информацию (т. е. заказ подтверждён и к нему добавлен налог с продаж). Безусловно, последовательное исполнение событий увеличивает время ожидания.

Можно, конечно, позволить сервисам подтверждения заказов и расчета налога работать одновременно, но в итоге мы получим два события с важной информацией в каждом из них: один подтверждённый заказ и один заказ с добавленным налогом с продаж. Таким образом, для получения правильного заказа с применением как подтверждения, так и налога с продаж необходимо объединить эти два сообщения. (В этом случае объединение должно происходить как в сервисе электронной почты, так и в сервисе представления заказа.)

В некоторых ситуациях такая возможность одновременно вносить изменения в одну и ту же сущность в разных процессах и объединять их позже может быть чрезвычайно мощной (например, совместное редактирование электронных документов). Но в других может приводить к ошибкам. И, как правило, при построении бизнес-систем, особенно связанных с деньгами (обработкой платежей и т. п.), разработчики склонны перестраховываться. Существует подход, который позволяет объединять данные, гарантируя при этом их целостность. Называется он «бесконфликтные реплицированные типы данных» (англ. conflict-free replicated data type, далее — CRDT).[109] Суть его заключается в ограничении круга операций, которые можно выполнять над данными, чтобы гарантировать,

что при их изменении и последующем объединении никакая информация не потеряется. Однако одним из минусов такого подхода является его ограниченная функциональность.

Хорошим компромиссом для крупных бизнес-систем является сохранение отставания (что позволяет иметь множество копий одного и того же состояния, доступных только для чтения) при полном исключении возможности конфликтов (путем запрета одновременных изменений). Чтобы этого добиться, нужно выделить единственного писателя (англ. single writer) для каждого типа данных (темы) или для каждого перехода состояния. Как раз об этом речь пойдёт далее.

Принцип единственного писателя

Принцип единственного писателя является полезным инструментом разрешения потенциальной несогласованности. Мартин Томпсон использовал этот термин,[110] критикуя широкое распространение механизмов блокировок записи в многопоточных системах (ведущее к падению производительности), и описывал альтернативное решение, когда операции записи может выполнять только один поток. Эта концепция тесно связана с «моделью акторов»[111] и некоторыми идеями, взятыми из исследований по базам данных[112] и проектированию систем. С точки зрения сервисов, этот принцип также сочетает в себе концепцию единой ответственности.[113]

Суть проста: ответственность за распространение событий определённого типа возлагается на один сервис — единственный писатель. Таким образом, сервис управления запасами отвечает за изменения количества товара на складе, сервис заказов отвечает за изменения заказов и т. д.

Объединение операций записи в один сервис упрощает эффективное управление согласованностью. Но преимущества этого принципа выходят далеко за рамки сохранения корректности данных при одновременном исполнении команд:

- Он позволяет применять контроль версий и проверку согласованности (например, проверку номера версии; см. «Идентификация и контроль параллелизма» на стр. 120) в одном месте.
- За логику изменений в каждом подразделении компании отвечает отдельный сервис, что облегчает обоснование и внедрение изменений (например, изменение схемы данных, как описано в Главе 13 «Управление изменениями в схеме данных и нарушение обратной совместимости» на стр. 136).
- За тот или иной набор данных отвечает тот или иной отдел, и

это позволяет его команде сосредоточиться только на своих задачах. Централизация схем данных и бизнес-логики могут препятствовать развитию системы.[114] Наглядный анти-пример, когда ESB и обмен сообщениями между различными системами реализованы через единый центр. Принцип единственного писателя предоставляет сервисам чётко определённые права на общие наборы данных, акцентируя внимание на внешних данных,[115] а также однозначно распределяя ответственность за них. Это особенно важно для отраслей, где есть сложные бизнес-правила, связанные с различными типами данных. Так, например, в сфере финансов, где для разработки и развития продуктов требуются глубокие знания предметной области, возможность закрепить ответственность за изменение данных за одним сервисом или отделом зачастую считается преимуществом.

Когда принцип единственного писателя применяется в сочетании с событийным сотрудничеством (см. Главу 5), каждый писатель изменяет часть единого бизнес-процесса через набор последовательных событий. На рис. 11.3 процесс работы интернет-магазина представлен более подробно. Несколько сервисов взаимодействуют друг с другом по мере того, как заказ проходит через различные стадии: создание, обработка платежа, доставка, завершение. Каждому сервису (заказа, платежей, доставки) соответствует своя тема. Каждый сервис контролируют все изменения состояний, произведённые в соответствующей теме.

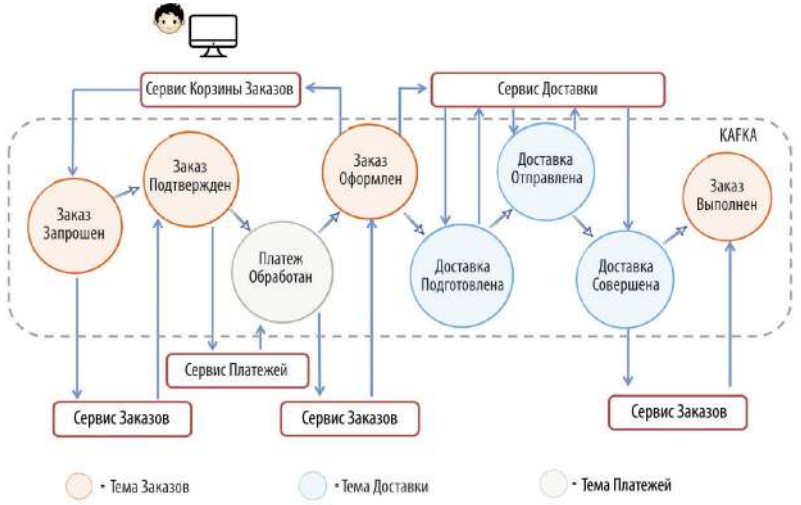


Рис. 11.3. Каждый круг представляет собой событие; цвет круга обозначает тему; процесс проходит от стадии «Заказ запрошен» до стадии «Заказ выполнен»; тем временем четыре сервиса производят разные изменения

состояний в тех темах, где они являются единственным писателем; общий бизнес-процесс задействует все четыре сервиса

Таким образом, вместо модели глобальной согласованности (взаимодействие через базу данных) мы используем принцип единственного писателя для создания локальных точек согласованности, которые связаны через поток событий. Существует несколько вариантов использования этого принципа, которые мы обсудим далее.

Как мы увидим в Главе 15 «Масштабировании параллельных операций в потоковых системах», можно линейно масштабировать единственных писателей с помощью секционирования, используя Kafka Streams API.

Управляющая тема

Распространённая модель применения этого принципа заключается в использовании двух тем для каждой сущности — Управляющая тема (англ. command topic) и тема Сущности (англ. entity topic). Это почти то же самое, что и обычное использование принципа единственного писателя. Отличие в том, что в управляющую тему может писать любой процесс, и эта тема используется только для исходного события. В тему Сущности может писать только главный сервис — единственный писатель. Выделение этих двух тем позволяет администраторам принудительно реализовать принцип единственного писателя через настройку разрешений для тем. Так, например, мы можем разделить события, связанные с заказом, на две темы, как показано в Таблице 11-1.

Таблица 11-1. Управляющая тема используется для отделения исходной инструкции от последующих событий

Тема	OrderCommandTopic	OrdersTopic
Типы	OrderRequest(ed)	OrderValidated, OrderCompleted
Писатель	Любой сервис	Сервис заказов

Единственный писатель для смены состояния

Менее строгий вариант принципа единственного писателя: сервисы отвечают за каждую отдельную смену состояния, а не за все изменения в теме (см. Таблицу 11-2). Так, например, сервис обработки платежа может вообще не использовать отдельную тему. Он может просто добавить необходимую информацию к существующему сообщению (в схеме данных объекта «Заказ» появится раздел «Платёж»). В таком случае сервис обработки платежа отвечает только за одну смену состояния, в то время

как за все остальные отвечает сервис заказов.

Таблица 11-2. Оба сервиса пишут сообщения в тему Заказов, но каждый сервис отвечает за разные смены состояний

Сервис	Сервис заказов	Сервис платежей
Тема	OrdersTopic	OrdersTopic
Записываемая смена состояния	OrderRequested->OrderValidated PaymentReceived->OrderConfirmed	OrderValidated->PaymentReceived

Атомарность транзакций

Kafka дает следующие гарантии на транзакции:

- Сообщения, отправляемые в рамках одной транзакции в разные темы, записываются все вместе, либо не записываются вообще.
- Сообщения, отправляемые в рамках одной транзакции в одну тему, никогда не будут дублироваться, даже в случае сбоя.

У транзакций в Kafka Streams есть одна очень интересная и полезная особенность: они позволяют атомарно объединять записи в хранилища состояний с записями в темы. Подробно транзакции в Kafka рассматриваются в Главе 12.

Идентификация и контроль параллелизма

Идентификация очень важна в бизнес-системах, но зачастую ей не уделяют должного внимания. Например, чтобы выявить дубликаты, сообщения должны быть однозначно идентифицируемы (Мы обсудим это в Главе 12). Идентификация также важна для управления одновременными обновлениями, если при этом реализуется оптимистичный контроль параллелизма.[116]

Желательно, чтобы идентификация соотносилась с хранимыми сущностями: у заказа должен быть OrderId, у платежа — PaymentId и т. д. Если у сущности могут быть логические изменения (например, у заказа есть несколько статусов: «Создан», «Подтверждён» и т. д., или покупатель может изменить адрес электронной почты), то у нее также должен быть номер версии:

```
"Customer"{
        "CustomerId": "1234"
        "Source": "ConfluentWebPortal"
        "Version": "1"
        ...
}
```

Номер версии можно использовать для управления параллельной обработкой. Например (рис. 11.4), покупатель по имени Боб открывает свой профиль в одной вкладке браузера (версия 1), а затем открывает ту же страницу в другой вкладке (опять версия 1). Затем он меняет свой адрес во второй вкладке. После этого сущности присваивается версия 2. Если Боб снова откроет первую вкладку и попытается изменить свой номер телефона, сервер должен сравнить версии и отклонить обновление.

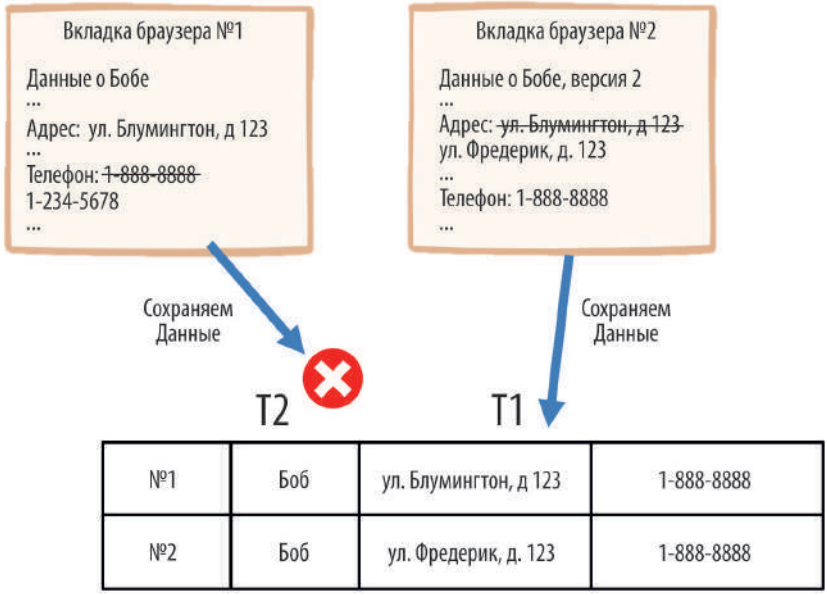


Рис. 11.4. Пример оптимистического контроля параллелизма. Сервер сравнивает номера версий прежде чем разрешить запись. Так как данные в первой вкладке устарели, запись транзакции T2 будет отклонена

Оптимистический контроль параллелизма можно реализовать как в синхронных, так и в асинхронных системах.

Ограничения

Принцип единственного писателя и другие модели, которые мы обсуждали в этой главе, работают в большинстве случаев, но всегда будут исключения. Например, при работе с устаревшими системами, где возможности изменений ограничены.

Заключение

В этой главе мы обсудили недостатки глобальной согласованности и преимущества итоговой согласованности. Затем адаптировали модель итоговой согласованности с применением принципа единственного писателя (сохраняя задержку в работе сервисов, но не допуская коллизий между ними). И, наконец, рассмотрели принципы, по которым реализуются системы идентификации и контроль параллелизма в событийно-ориентированных системах.

ГЛАВА 12

Новый взгляд на транзакции

Аналогично большинству реляционных баз данных Kafka поддерживает транзакции. Их реализация, как мы увидим далее, значительно отличается, но цель похожа и заключается в том, чтобы результаты, которые выдают программы, были предсказуемы и воспроизводимы, даже в случае сбоев.

С точки зрения сервисов, у транзакций есть три важных свойства:

- Удаление дублей, из-за которых многие потоковые операции (даже такие простые, как подсчет количества записей) дают неверные результаты.
- Атомарная запись пакетов сообщений в разные темы, например, «Заказ оформлен» и «Уменьшение количества товара на складе». Это исключает ситуацию, когда сбой при отправке одного из этих сообщений приводит к несогласованному состоянию системы.
- Атомарная запись событий и состояний. В Kafka Streams используются хранилища состояний, которые опираются на темы. Поэтому, когда мы сохраняем данные в хранилище состояний, а затем отправляем сообщение другому сервису, можно инкапсулировать обе операции в одну транзакцию. Данное свойство является особенно полезным.

В этой главе мы рассмотрим транзакции более подробно: узнаем, какие проблемы можно решать с их помощью, как использовать транзакции, и в чем заключается принцип их работы.

Проблема дублирования

Любая сервис-ориентированная архитектура представляет собой распределённую систему. Такие системы известны своей сложностью,

особенно заметной, когда что-то идет не так. Различные мысленные эксперименты и теоретические доказательства (например, «Задача о двух генералах»[117] или теорема Фишер-Линч-Патерсон[118] о невозможности консенсуса в системе со сбоями) в очередной раз подчёркивают неизбежность трудностей, связанных с работой распределённых систем. Но на практике эта проблема кажется менее сложной. Если при обращении к сервису он по какой-то причине «не отвечает», то попытка повторяется, и в конечном счете вызов будет выполнен.

Тут, правда, есть один недостаток: повторные вызовы могут привести к повторной обработке того же запроса, а это чревато вполне реальными проблемами. Например, повторное списание средств со счёта пользователя приведёт к некорректному отображению остатка (рис. 12.1). Добавление повторяющихся сообщений в ленту испортит впечатление пользователя от взаимодействия с системой. Список можно продолжать.

Рис. 12.1. Пользовательский интерфейс (UI) обращается к сервису платежей. Тот, в свою очередь, обращается к внешней платёжной системе. Прежде чем UI успевает получить ответ от платёжной системы, в работе последнего происходит сбой. Поскольку время ожидания закончилось, а UI так и не получил ответа, он отправляет повторный запрос. Происходит повторное списание средств со счёта пользователя

В реальной жизни большинство систем обрабатывает дубли автоматически: данные направляются в базу данных, которая автоматически производит дедупликацию, т. е. удаление избыточных данных, по первичному ключу. Такие процессы являются идемпотентными.[119] Таким образом, если пользователь обновляет свой адрес, и эта информация сохраняется в базе данных, нам неважно, создаются ли при этом дубли. Худшее, что может случиться, — таблица, в которой хранятся адреса пользователей, обновится дважды. Ничего страшного в этом нет. То же самое справедливо для примера с обработкой платежа, при условии, что у каждого отправленного запроса есть уникальный идентификатор. Поскольку дедупликация происходит в конце каждого упомянутого сценария, не имеет значения, сколько всего повторяющихся вызовов было выполнено. На самом деле, это довольно старая идея, которая появилась еще на ранних этапах разработки протокола TCP. Называется это «принцип прозрачности» (англ. end-to-end principle).[120]

Загвоздка вот в чем: чтобы такая естественная дедупликация работала, нужно, чтобы каждый вызов по сети:

- имел соответствующий идентификационный ключ;
- был дедуплицирован в базе данных, которая хранит историю всех ключей за заданный период. Или же дубликаты должны постоянно учитываться в бизнес-логике, которую мы прописываем, что увеличивает архитектурную сложность этой задачи.

Событийно-ориентированные системы стараются отходить от такого стиля обработки информации, ориентированного на обработку в базе данных. Вместо этого они применяют бизнес-логику к данным и передают результаты их обработки дальше.

В результате большинство событийно-ориентированных систем при получении сообщений производят сначала их дедупликацию, а затем уже обработку. Каждому отправленному сообщению, в свою очередь, присваивается тщательно подобранный идентификатор, по которому впоследствии можно так же удалить дубли. В лучшем случае вы столкнётесь с небольшими трудностями. В худшем — стоит ожидать вал ошибок.

Но, если подумать, дедупликация не является проблемой уровня приложений, точно так же, как упорядочение сообщений, организация их повторной отправки и другие полезные функции TCP. Мы отдаём предпочтение именно TCP (а не UDP, англ. user datagram protocol), так как этот протокол позволяет писать программы на более высоком уровне абстракции, где о доставке, упорядочении сообщений и т. д. уже позаботились за нас. Остаётся только догадываться, почему же борьба с дублями просочилась на уровень приложений. Разве эта проблема не должна решаться на уровне инфраструктуры?

Транзакции в Kafka позволяют создавать длинные цепочки сервисов, где обработка каждого шага происходит строго один раз. В результате уменьшается количество дублей, а значит, сервисы становится легче программировать. Кроме того, как мы увидим далее в этой главе, транзакции позволяют объединять потоки событий и состояния при использовании хранилища состояний Kafka Streams или подхода «источники событий». Если вы используете Kafka Streams API, то всё это происходит автоматически.

Но, не нужно думать, что это какое-то волшебное средство, которое гарантирует доставку сообщений строго один раз (англ. exactly once) в рамках всей вашей системы. В конечном счете она состоит из множества разных частей. Некоторые из них работают на Kafka, а некоторые — на других технологиях, на которые гарантия доставки строго один раз не распространяется.

Хотя, в случае с Kafka, это и правда напоминает волшебство. Все взаимодействия между сервисами записываются строго один раз (рис. 12.2). Это освобождает сервисы от необходимости дедупликации входящих данных и выбора соответствующих ключей для исходящих данных. Таким образом, мы можем объединить сервисы в один событийно-ориентированный процесс без каких-либо дополнительных проблем. Это очень удобно.

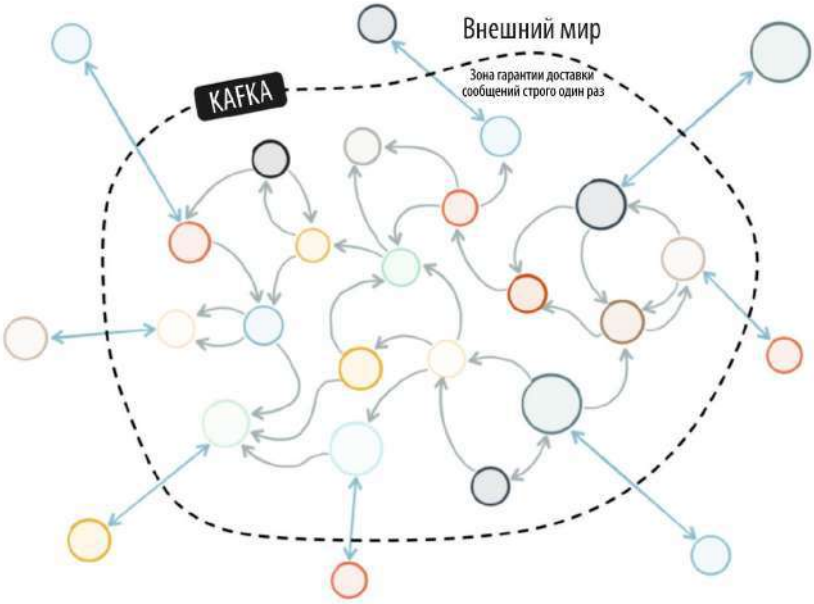


Рис. 12.2. Транзакции Kafka обеспечивают гарантии только для тех взаимодействий, которые происходят внутри Kafka

Использование транзакционных API для удаления дублей

В качестве простого примера представьте, что у нас есть сервис подтверждения состояния счёта. Он принимает сообщение о поступлении средств на счёт, проводит проверку сообщения, а затем отправляет новое сообщение обратно в Kafka о том, что поступление подтверждено.

Kafka записывает прогресс чтения каждого покупателя, сохраняя указатель на последнее прочитанное сообщение (англ. offset) в специальную тему `consumer_offsets`. Таким образом, чтобы подтвердить каждое поступление на счёт строго один раз, оба действия — (a) отправка сообщения «Подтвердить внесение» обратно в Kafka и (b) фиксация номера сообщения в

теме consumer_offsets — должны рассматриваться как единая атомарная единица изменения (рис. 12.3). Ниже приведен пример кода, реализующего данную логику:

```
//Чтение и подтверждение поступлений
validatedDeposits = validate(consumer.poll(0))

//Атомарная отправка сообщения «Подтвердить внесение» и
фиксация указателя на последнее прочитанное сообщение
producer.beginTransaction()
producer.send(validatedDeposits)
producer.sendOffsetsToTransaction(offsets(consumer))
producer.endTransaction()
```

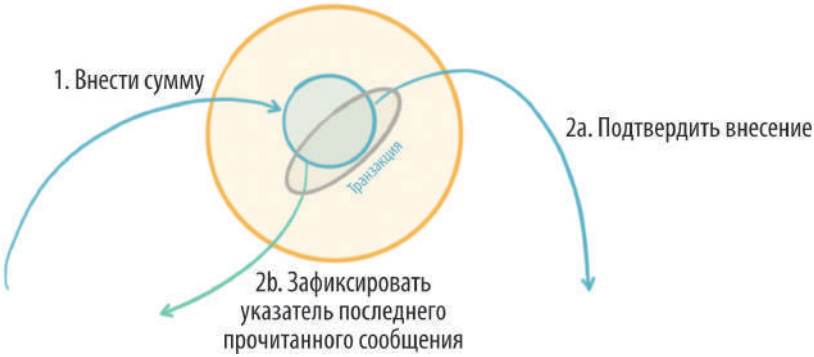


Рис. 12.3. Отправка сообщения фактически разделяется на две операции: отправка самого сообщения и подтверждение отправки. При этом во избежание дублей обе операции должны выполняться атомарно

Если вы используете Kafka Streams API, не нужно писать дополнительный код. Все уже работает.

Доставка строго один раз — это одновременно и идемпотентность, и атомарная фиксация изменений

Поскольку Kafka является брокером сообщений, дубликаты могут появляться двумя способами. Во-первых, сбой может произойти при отправке сообщения в Kafka. Если издатель не получит подтверждение записи, он предпримет еще одну попытку, что может привести к дублированию сообщений. Во-вторых, сбой может произойти при чтении сообщения из Kafka. Если чтение (т. е. указатель последнего прочитанного сообщения) не было зафиксировано, сообщение может быть прочитано

повторно, когда процесс-подписчик восстановится (рис. 12.4).

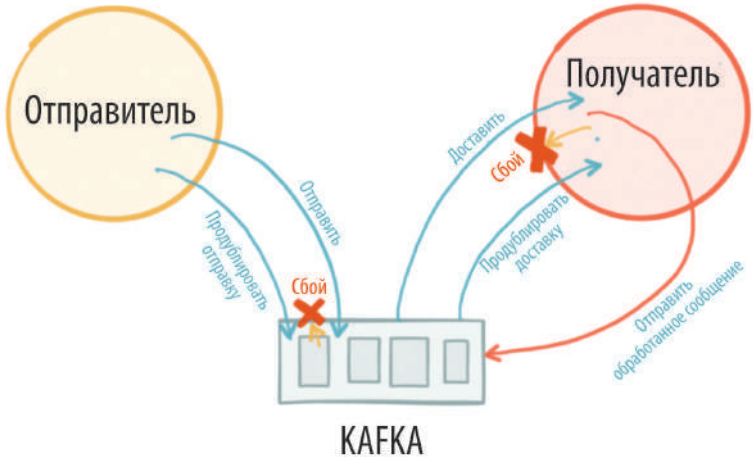


Рис. 12.4. У брокеров сообщений есть два сценария сбоя: во время записи и во время чтения

Таким образом, чтобы исключить возможность создания дублей в журнале сообщений, брокерам сообщений необходимо обладать свойством идемпотентности. В данном контексте идемпотентность — то же самое, что дедупликация. У каждого издателя есть идентификатор, а у каждого сообщения — порядковый номер. Их комбинация позволяет однозначно идентифицировать каждый отправленный пакет сообщений. Брокер использует этот уникальный порядковый номер, чтобы определить, было ли сообщение уже записано в журнал, и, если да, удалить его. Это более эффективный подход, чем хранение всех когда-либо созданных ключей в базе данных.

На «читающей» стороне можно было бы просто дедуплицировать сообщения (например, в базе данных). Однако Kafka дает более широкие гарантии на транзакции, более схожие с гарантиями базы данных. Транзакции Kafka связывают сообщения, отправленные вместе, в одну атомарную фиксацию изменений. Таким образом, брокер сначала реализует идемпотентность, а поверх нее — атомарную фиксацию.

Как работают транзакции в Kafka

Как вы могли заметить из примера кода в предыдущем пункте, реализация транзакций в Kafka очень похожа на их реализацию в базе данных. Вы начинаете транзакцию, производите запись сообщений в Kafka, затем либо фиксируете транзакцию, либо отменяете. Однако модель транзакций в Kafka кардинально отличается, потому что она разработана для потоковой

передачи данных.

Одно из ключевых различий — использование сообщений-маркеров, которые могут применяться в разных потоках. Идея маркеров впервые была сформулирована в рамках алгоритма Чанди-Лампорта почти 30 лет назад.[121] Транзакции в Kafka являются адаптацией этой идеи, хотя и преследуют несколько иную цель.

Несмотря на то, что такой подход к транзакциям сложно реализовать на практике, с точки зрения понимания всё достаточно просто (рис. 12.5). Возьмём предыдущий пример, где два сообщения атомарно записывались в две разные темы. Одно сообщение было записано в тему Внесения, другое — в тему committed_offsets.

Сначала к обеим темам отправляются маркеры Начала транзакции.[i] Далее идут непосредственно сообщения. И, наконец, когда все сообщения отправлены, транзакция заканчивается одним из двух маркеров: Фиксация или Отмена.

Концепция маркеров Начала транзакции тоже адаптирована под транзакции. Цель транзакции — убедиться, что подписчики видят только подтверждённые данные. Это работает следующим образом. При получении маркера Начала транзакции подписчик начинает процесс внутренней буферизации. Отправленные сообщения не записываются сразу, а дожидаются маркера Фиксации транзакции. Тогда и только тогда сообщения становятся доступны подписчику. Благодаря такой буферизации подписчики читают только подтверждённые сообщения.

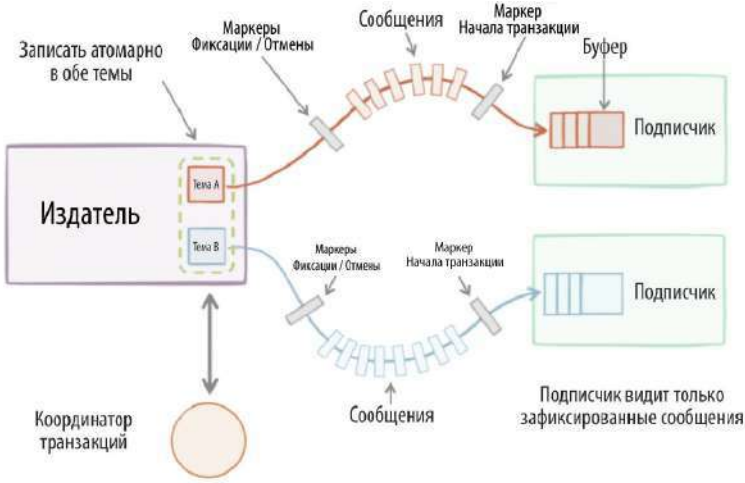


Рис. 12.5. Концептуальная модель транзакций в Kafka

i На практике в целях оптимизации задача буферизации переходит от брокера к подписчику, что позволяет снизить нагрузку на память.

Для обеспечения атомарности каждой транзакции отправка маркеров Фиксации происходит с помощью координатора транзакций. Таких координаторов в кластере может быть много, так что, если на одном из них произойдет сбой, система продолжит функционировать. Тем не менее, для каждой транзакции используется только один координатор.

Координатор транзакций является последней инстанцией. Он отмечает, что транзакция была зафиксирована и сохраняет это состояние в журнале транзакций (двухфазная фиксация[122]).

Безусловно, двухфазная фиксация влечёт за собой снижение производительности системы, которое было бы действительно заметно при необходимости фиксировать каждое сообщение. Но на практике такой необходимости нет. Фиксируются не отдельные сообщения, а пакеты сообщений, благодаря чему достигается баланс между производительностью и задержкой. Например, пакеты сообщений размером 1 КБ, фиксация которых происходит каждые 100 мс, увеличивают общее время обработки данных на 3% по сравнению с доставкой сообщений по порядку с семантикой доставки хотя бы один раз (англ. at-least-once).[123] Вы можете сами это проверить с помощью встроенных скриптов тестирования производительности Kafka.

В реализации транзакций на практике есть много тонкостей, таких как восстановление после сбоя, избавление от зомби-процессов и правильное присвоение идентификаторов. Тем не менее, той модели, что мы рассмотрели, достаточно для общего понимания их работы. Если вас интересует более подробный разбор принципов работы транзакций, читайте в блоге Confluent.[124]

Атомарная запись состояний и событий

В Главе 7 мы говорили о том, что данные можно сохранять в журнал событий Kafka. Обычно для этого используется хранилище состояний (хэш-таблица на жёстком диске внутри API-интерфейса, привязанная к темам Kafka) в Kafka Streams. Поскольку устойчивость хранилища состояний в Kafka привязана к темам (изменения не могут быть потеряны даже в случае сбоев), можно использовать транзакции, чтобы объединить записи хранилища состояний с записями в темы. Это очень эффективный подход, поскольку он имитирует процесс слияния сообщений и данных в атомарные транзакции. Этот же процесс в обычных базах данных выполняется через протокол типа XA[125] и происходит мучительно долго.

Дополним наш прошлый пример. Теперь сервис подтверждения поступлений фиксирует баланс в соответствии с внесением средств. Таким образом, если начальный баланс равен 50 $, и на счёт поступает еще

Примечание:

База данных в Kafka Streams — это хранилище состояний. Поскольку хранилища состояний «берут» данные из тем Kafka, транзакции позволяют атомарно связывать отправляемые сообщения и сохраняемое состояние.

5 $, то баланс на счёте должен стать равным 55 $. Теперь мы не только фиксируем, что на счёт поступило 5 $, но и сохраняем текущее значение баланса 55 $ в хранилище состояний (или напрямую в уплотнённую тему).

Рис. 12.6. Атомарная отправка трёх сообщений: внесение средств, текущее значение баланса, фиксация номера сообщения

Если в Kafka Streams включена функция транзакций, все эти операции будут автоматически объединены в транзакцию. В результате баланс на счёте всегда будет атомарно синхронизироваться с внесениями средств. Такие же процессы можно реализовать для подписчиков: можно выводить количество товара на складе (если вручную инкапсулировать такой запрос в транзакцию[126]) или баланс счёта (если при каждом входе в систему запрашивать его баланс).

В этом примере важно то, что в нём сочетаются концепции обмена сообщениями и управления состоянием. Мы не только реагируем на события, обрабатываем их и создаем новые события, но также управляем состоянием (балансом счёта) в Kafka — всё это в рамках одной транзакции.

Нужны ли нам транзакции? Можно ли добиться того же результата с помощью идемпотентности?

Разработчики десятилетиями создавали системы, ориентированные как на события, так и на взаимодействие «запрос-ответ», добиваясь

идемпотентности процессов с помощью идентификаторов и баз данных. Однако реализация идемпотентности сопряжена с определёнными проблемами. Безусловно, определение идентификатора заказа — задача несложная. Тем не менее, в других потоках событий с этим могут возникать проблемы. Возьмём, к примеру, поток событий, представляющий собой средний баланс счёта по регионам в час. Конечно, мы могли бы подобрать подходящий ключ, но представьте себе, насколько он будет ненадёжным.

Кроме того, транзакции полностью инкапсулируют процесс дедупликации внутри сервиса. Вы не засоряете «эфир» сервисов-подписчиков дублями, а алгоритм работы каждого сервиса становится более простым и понятным. Реализация идемпотентности, в свою очередь, требует от каждого сервиса-подписчика корректной дедупликации получаемых данных, что естественным образом усложняет их алгоритм работы и вынуждает их чаще ошибаться.

Чего не могут делать транзакции?

У транзакций есть некоторые ограничения. Кому-то они могут показаться очевидными, но всё же стоит их упомянуть. Во-первых, транзакции работают только тогда, когда и ввод, и вывод данных происходят через Kafka. Транзакционные гарантии распространяются только на запись в брокер Kafka и чтение из него. При вызове внешнего сервиса (например, через протокол HTTP), обновлении базы данных, записи в стандартный вывод `stdout` или совершении других операций вызовы могут дублироваться. Таким образом, как и при использовании транзакционной базы данных, транзакции работают только тогда, когда вы используете Kafka.

Как и в случае с базой данных, транзакции фиксируются при отправке сообщений. Как только транзакция была зафиксирована, её невозможно отменить, даже если во время следующей транзакции произойдет сбой. Допустим, UI отправляет транзакционное сообщение сервису заказов. Затем, при отправке собственных сообщений у сервиса заказов происходит сбой. Тогда все сообщения, отправленные сервисом заказов, будут отменены. При этом отменить транзакцию, отправленную через UI, невозможно. Если вам нужны мультисервисные транзакции, рассмотрите использование шаблонов Сага (англ. Saga).[127]

Транзакции записываются в брокер атомарно (так же, как и в базу данных). Однако нет гарантий относительно того, когда тот или иной подписчик прочитает эти сообщения. Это может показаться очевидным, но иногда приводит к путанице. Допустим, мы отправляем два сообщения — в тему Заказы и в тему Платежи — в рамках одной транзакции. Мы не знаем, когда подписчик прочитает одно или второе сообщение, и прочитает ли он их

одновременно. В транзакционной базе данных происходит то же самое.

И, наконец, в перечисленных примерах мы используем API издателя и подписчика, чтобы продемонстрировать, как работают транзакции. Но если вы используете Kafka Streams API, то никакой дополнительный код не требуется. Просто примените соответствующие настройки, и семантика доставки строго один раз начнёт работать автоматически.

На данный момент транзакции полностью поддерживаются только для отдельных издателей и подписчиков, но не для групп подписчиков (сейчас эта функциональность в разработке). Если вам необходимо работать именно с группами подписчиков, используйте Kafka Streams API, там эта функциональность реализована полноценно.

Использование транзакций в сервисах

В Главе 5 мы говорили о модели, известной как событийное сотрудничество — сообщения идут от сервиса к сервису, создавая единый процесс. Он начинается с события «Заказ запрошен» и заканчивается событием «Заказ выполнен». Посередине задействованы ещё несколько сервисов, которые способствуют обработке заказа.
Транзакции особенно важны в таких сложных рабочих процессах, где трудно применить принцип прозрачности. Без них дедупликация должна была бы происходить в каждом сервисе. Кроме того, было бы довольно сложно создать надёжное потоковое приложение без использования транзакций. На это есть несколько причин:

а) Потоковые приложения используют много промежуточных тем. Их дедупликация после каждого шага была бы очень обременительной (а в KSQL почти невозможной).
б) Языки DSL позволяют через одну операцию вызывать несколько методов (например, `flatMap()`), а это усложняет управление идемпотентностью без API для транзакций.

Транзакций в Kafka решают эти проблемы, а также атомарно связывают потоковую обработку с хранением промежуточных состояний в хранилищах состояний.

Заключение

Транзакции влияют на то, как мы разрабатываем сервисы:

- Они освобождают сервисы, связанные с Kafka, от необходимости гарантировать идемпотентность. При разработке сервисов,

работающих по шаблону «чтение, обработка, (сохранение,) запись», не нужно беспокоиться о дедупликации входящих сообщений или создании ключей для исходящих сообщений.
- Больше не нужно беспокоиться о том, что отправляемые сообщения должны иметь соответствующие уникальные идентификаторы. Данное преимущество в меньшей степени относится к темам, которые содержат бизнес-события — обычно там и так есть надежные ключи. Однако это свойство очень полезно, когда мы управляем производными / промежуточными данными, например, когда мы перераспределяем события, создаем агрегированные события или используем Streams API.
- Если требуется отказоустойчивость, то в Kafka можно объединять сообщения, отправляемые другим сервисам, и состояние, которое нужно сохранить, в одну транзакцию, которая будет либо зафиксирована, либо отменена. Это облегчает создание простых приложений и сервисов с состоянием.

Проще говоря, когда вы создаете событийно-ориентированные системы, транзакции Kafka освобождают вас от забот о сбоях и повторных вызовах — задача, которую на самом деле должна решать инфраструктура, а не код. Это повышает уровень абстракции, облегчая получение точных, повторяемых результатов от большого массива микросервисов.

Однако, нужно быть осторожными. Транзакции решают только одну из проблем распределённых систем, но есть много других. Комплексные сервисы по-прежнему необходимы. Тем не менее, в мире, где ценится скорость и гибкость, потоковые платформы ещё выше поднимают планку, позволяя нам создавать больше разноплановых микросервисов, которые будут вести себя так же предсказуемо в сложных цепочках, как и по отдельности.

ГЛАВА 13

Изменение данных и схем с течением времени

Схемы — это API-интерфейсы, используемые событийно-ориентированными сервисами. Таким образом, издатель и подписчик должны согласовать между собой формат передаваемого сообщения. В результате между ними возникает логическая связь, так как они используют общую схему данных. Аналогично тому, как сервисы, работающие в парадигме «запрос-ответ», используют технологии обнаружения сервисов для поиска доступных API-интерфейсов, событийно-ориентированные сервисы нуждаются в механизме обнаружения доступных тем и описания данных (т. е. схем), которые они предоставляют.

Существует достаточно много стандартов описания схем. Среди наиболее популярных — Protobuf[128] и JSON Schema.[129] Однако для реализации большинства проектов в Kafka используется Avro.[130] Для централизованного управления и верификации схем Confluent предлагает Schema Registry[131] — решение на базе открытого кода, которое представляет собой централизованный репозиторий для схем Avro.

Управление изменением данных с помощью схем

Схемы данных представляют собой соглашение, которое определяет, как должно выглядеть сообщение. Это интуитивно понятно. Важно, однако, что в большинстве форматов схем предусмотрен механизм проверки обратной совместимости сообщения, записанного по новой схеме, с предыдущими версиями (или наоборот). Это свойство является необходимым. (Не используйте сериализацию Java или любой другой неизменяемый формат в сервисах, в которых форматы могут меняться независимо.)

К примеру, вы добавили поле «код возврата» для схемы заказа. Это —

изменение с обратной совместимостью. Программы, работающие со старой схемой, всё ещё смогут прочитать сообщение, но не будут видеть поле «код возврата» (прямая совместимость). Программы с новой схемой смогут читать сообщение целиком, включая «код возврата» (обратная совместимость).

К сожалению, нельзя обеспечить совместимость при перемещении или удалении полей из схемы. Однако обычно можно реализовать такие изменения, создав клон схемы. В этом случае данные будут дублироваться в двух местах до тех пор, пока не произойдёт полная миграция.

Возможность вносить в схемы изменения, добавляя новые поля, которые не нарушают работу старых программ, лежит в основе управления большинством систем обмена сообщениями.

Можно использовать Schema Registry для контроля этой совместимости. Schema Registry сопоставляет темы Kafka со схемами, которые они используют (рис. 13.1), а также обеспечивает соблюдение правил совместимости прежде, чем сообщения будут записаны в тему. Таким образом, Schema Registry проверяет каждое сообщение, отправленное в Kafka, на предмет совместимости со схемой Avro, гарантируя тем самым, что несовместимые сообщения опубликованы не будут.

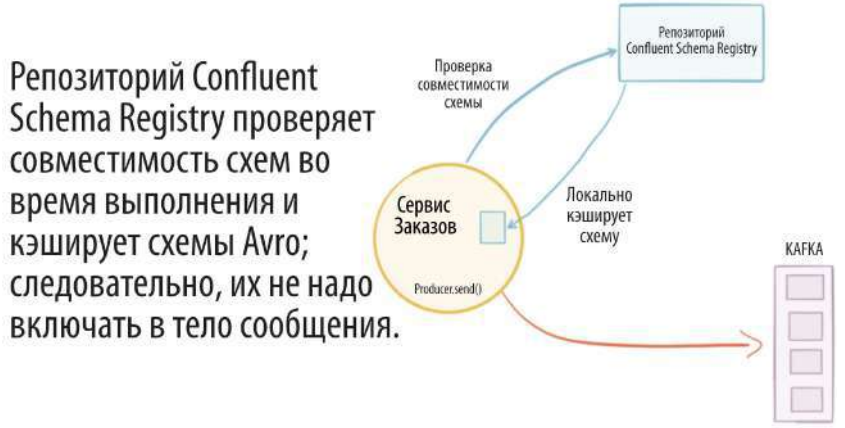


Рис. 13.1. Обращение к Schema Registry для проверки совместимости схемы при чтении и записи сообщений в сервис заказов

Управление изменениями в схеме данных и нарушение обратной совместимости

Длительная миграция схем является одним из наиболее частых объектов критики в адрес реляционных систем. Но реальность такова, что изменение

схем является неотъемлемой частью процесса устаревания данных. Если раньше использовались только схемы на запись, то сейчас появились схемы на чтение[132] (данные сначала записываются, а схема добавляется позже, при чтении). Они позволяют множеству несовместимых схем данных сосуществовать в одной и той же таблице, теме и т. д. В результате проблема преобразования формата от старого к новому выходит на уровень приложений, отсюда и название «схема на чтение».

У такой схемы есть несколько преимуществ. Чаще всего новые данные являются более ценными, чем старые. При использовании схемы на чтение у программ нет необходимости мигрировать старые данные, которые им не очень-то нужны. Это полезное, прагматичное решение, которое широко используется на практике, особенно в системах обмена сообщениями. Кроме того, схема на чтение может быть проста в реализации, если код для работы со старыми схемами уже существует в кодовой базе (на практике зачастую так и есть).

Однако какой подход вы бы ни выбрали, вам никуда не деться от необходимости решать проблемы, связанные с изменением схем (пожалуй, единственное исключение — единое, глобальное обновление всей системы), будь то физическая миграция наборов данных или изменение структуры данных в логике приложения. Kafka в этом смысле ничем не отличается.

Как мы уже упомянули в предыдущем разделе, чаще всего обратная совместимость между схемами поддерживается благодаря реализации аддитивных изменений (т. е. поля можно добавлять, но нельзя перемещать или удалять). Однако время от времени схемы нуждаются в «несовместимых» обновлениях. Наиболее распространённый подход к реализации таких обновлений — параллельное использование двух схем, которое дает сервисам окно времени для обновления (англ. dual schema upgrade window). В рамках данного подхода создаются две темы — orders-v1 и orders-v2 — для сообщений, записанных по старой и новой схемам, соответственно. Предположим, что в тему Заказы пишет только сервис заказов. Тогда мы имеем несколько вариантов:

- Сервис заказов записывает сообщения одновременно по обеим схемам в обе темы, используя API транзакций Kafka, чтобы гарантировать атомарность записи. (При таком подходе миграция не распространяется на исторические данные, поэтому не подходит для тем, которые используются для долгосрочного хранения информации.)
- Сервис заказов «перенаправляется» и начинает записывать сообщения только в тему orders-v2. В Kafka Streams в таком случае добавляется задание на конвертацию сообщений из orders-v2 в orders-v1 для обеспечения обратной совместимости. (В данном

подходе миграция также не распространяется на исторические данные.) См. рис. 13.2.

- Сервис заказов продолжает записывать сообщения в тему orders-v1. Задача Kafka Streams в таком случае заключается в конвертации сообщений из orders-v1 в orders-v2 до тех пор, пока все клиенты не обновятся. Когда это произойдет, сервис заказов будет перенаправлен к теме orders-v2. (Здесь миграция уже распространяется и на исторические данные.)
- Сервис заказов производит внутреннюю миграцию данных (в собственной базе данных), затем перезаписывает весь массив сообщений в тему orders-v2. Далее он продолжает писать в обе темы — orders-v1 и orders-v2, используя соответствующие форматы. (Миграция распространяется на исторические данные.)

Все четыре подхода позволяют достичь одной цели: предоставить сервисам окно времени для обновления. Два последних подхода облегчают процесс перемещения старых сообщений из orders-v1 в orders-v2. Kafka Streams делает это автоматически, если миграция начинается с самого первого сообщения. Такая реализация больше подходит для тем с долгим удержанием данных, например, если Kafka используется в качестве источника событий.

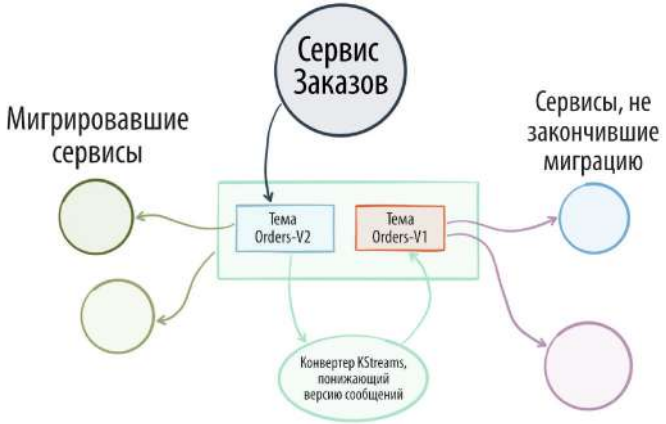


Рис. 13.2. Окно времени для обновления: одни и те же данные сосуществуют в двух темах с разными схемами, предоставляя сервисам временя для обновления

Сервисы продолжают использовать две темы до тех пор, пока не произойдет полная миграция в тему orders-v2. Тогда тему orders-v1 можно будет архивировать или удалить.

Кстати говоря, одна из причин для использования принципа единственного

писателя, о котором мы говорили в Главе 11, заключается как раз в упрощении процесса обновления схем. Если бы в тему Заказы писало сразу несколько разных сервисов, было бы намного сложнее запланировать обновление каждого из них в отдельности без обратной совместимости.

Совместная работа над изменением схемы

В предыдущем разделе мы рассмотрели реализацию изменений схемы без обратной совместимости. Стоит отметить, что на практике реализация даже самого незначительного изменения обычно требует межкомандного взаимодействия, в ходе которого определяется целесообразность данного изменения. Такое взаимодействие может происходить разными способами. Один из наиболее распространенных вариантов заключается в оповещении заинтересованных команд по электронной почте с указанием сути изменений и срока их вступления в силу. Другой вариант — когда одна команда отвечает за координацию всего процесса миграции. Тем не менее, оба подхода не очень хорошо работают на практике. В первом случае (с рассылкой сообщений) не хватает чёткой структуры организации работы и распределения ответственности всех участников. Второй подход (с единой командой) тормозит процесс, так как разработчикам приходится ждать пока ответственная команда реализует изменения и затем даст отмашку, что можно использовать новую схему.

Более удачный способ совместной реализации изменений — использование GitHub. На то есть две причины:

а) схемы — это такой же код, и поэтому к ним так же должен применяться контроль версий;
б) GitHub позволяет разработчикам предлагать изменения и создавать запрос на предложение своих изменений (англ. pull request, далее — PR) в репозиторий. Все будут заранее в курсе грядущих изменений и смогут учесть их при разработке и тестировании системы.

Заинтересованные лица могут просматривать, комментировать и утверждать изменения. Как только будет достигнут консенсус, можно объединять PR с репозиторием и разворачивать новую схему. Этот подход является наиболее оптимальным, поскольку обеспечивает надёжную процедуру достижения (и проверки) консенсуса относительно того или иного изменения, не создавая при этом лишних препятствий для дальнейшей разработки.

Обработка нечитаемых сообщений

Наличие схем не всегда гарантирует, что нижестоящие приложения смогут без проблем читать данные. Всегда может появиться какая-нибудь семантическая ошибка, например неверный символ, недопустимый код страны, отрицательное значение или даже некорректные байты из-за повреждения данных, и нарушить весь процесс. Обычно при возникновении таких ошибок процесс обработки данных приостанавливается до тех пор, пока проблема не будет решена. Однако в некоторых средах такое неприемлемо.

В традиционных системах обмена сообщениями обычно есть так называемая очередь недоставленных сообщений (англ. dead letter queue).[133] Туда поступают сообщения, которые не могут быть отправлены, например, из-за того, что их не удаётся поместить в целевую очередь. В Kafka такой очереди нет, однако могут возникать ситуации, когда подписчик не может прочитать сообщение из-за семантических ошибок или некорректности данных внутри сообщения (например, ошибка контрольной суммы при чтении из-за того, что данные были повреждены).[134]

Некоторые разработчики предпочитают создавать что-то похожее на очередь недоставленных сообщений в отдельной теме. Если по какой-то причине подписчик не может прочитать сообщение, оно помещается в очередь для сообщений с ошибками, чтобы не тормозить весь процесс обработки данных. Позже сообщения из этой очереди можно обработать повторно.[135]

Удаление данных

Когда данные в журнале сообщений хранятся в течение длительного или даже неопределённого периода времени, иногда возникает необходимость удалять сообщения, исправлять ошибки или повреждённые данные, а также редактировать разделы, содержащие конфиденциальные данные. Хороший тому пример — принятие Евросоюзом Общего регламента по защите данных (англ. General Data Protection Regulation, далее — GDPR), который, среди прочего, предоставляет пользователям «право на забвение» (полное удаление данных о человеке из баз данных компании).

Самый простой способ удалить сообщения из Kafka — просто дождаться, когда истечёт срок их хранения. По умолчанию Kafka хранит данные в течение двух недель, однако этот параметр можно менять как в большую, так и в меньшую сторону. Существует также API-интерфейс системного администратора, который позволяет напрямую удалять сообщения, которые устарели сами или для которых устарел указатель о чтении (offset).

При использовании Kafka в качестве источника событий или источника истины, обычно не требуется удалять сообщения. Удаление происходит путём замены записи на пустое значение (англ. null) (или маркер удаления при необходимости). Таким образом, история всех предыдущих версий объекта сохраняется. Большинство приёмников данных Kafka Connect разрабатываются именно с учётом маркеров удаления.

Но для удовлетворения требований таких регламентов, как GDPR, подобного маркера будет недостаточно, поскольку все данные должны быть физически удалены из системы. Есть несколько вариантов решения этой проблемы. Некоторые подходят к этому вопросу с позиций безопасности и используют, например, криптографическое измельчение[136] — удаление или перезапись ключей шифрования. Однако большинство предпочитает использовать уплотнённые темы, поскольку они позволяют напрямую удалять или заменять сообщения на основании их ключей.

Стоит отметить, что данные из уплотнённых тем удаляются иначе, чем из реляционной базы данных. Kafka использует механизм, более похожий на тот, что применяют Cassandra и HBase: записи помечаются для удаления, а позже, при запуске процесса уплотнения, удаляются. Удалить сообщение из уплотнённой темы очень просто: нужно записать новое сообщение в тему, указав ключ, который нужно удалить, и пустое значение (null). Когда запустится процесс уплотнения, сообщение будет удалено навсегда.

Если в качестве ключа темы используется не CustomerId, а другой идентификатор, то нужно их как-то сопоставить. Например, если мы рассматриваем тему Заказы, то нужен какой-то процесс, который сопоставит CustomerId и OrderId. Затем, чтобы «забыть» пользователя, нужно просто найти все его заказы и либо совсем удалить их из Kafka, либо убрать из них всю информацию об этом пользователе. Это можно сделать самостоятельно или с помощью Kafka Streams.

Менее распространенная ситуация, которую тоже стоит упомянуть, возникает, когда ключ темы Kafka не связан явным образом с тем ключом, по которому вы хотите произвести удаление. Предположим, нужно выбрать заказы по ProductId. Удалить заказы отдельных пользователей по такому ключу не получится, так что простой способ, описанный выше, не сработает. Однако есть и другой вариант — создать составной ключ из [ProductId] и [CustomerId], а затем на стороне издателя реализовать код, который будет вычислять ProductId и отделять сообщения только по этому ключу. Теперь вы можете удалить сообщения по тому же принципу, что мы обсудили ранее, используя пару [ProductId][CustomerId] в качестве ключа.

Запуск удалений в нижестоящих базах данных

Зачастую процесс разработки построен таким образом, что Kafka перемещает данные из одной базы данных в другую с помощью коннекторов. В этом случае вам понадобится удалить запись в исходной базе данных, а затем распространить это изменение через Kafka на все приёмники данных Kafka Connect. Если вы используете функцию отслеживания измененных данных (CDC), она сделает всю работу за вас: коннектор источника данных зафиксирует удаление и через поток событий Kafka распространит его на приёмники данных. Если вы не используете коннектор с поддержкой CDC, вам понадобится разработать собственный механизм для управления удалениями.

Разделение публичных и частных тем

При использовании Kafka в качестве источника событий или для потоковой обработки данных и наличии единственного кластера, через который взаимодействуют различные сервисы, логично было бы отделить частные, внутренние темы от общих бизнес-тем.

Некоторые команды предпочитают в этом вопросе действовать по договорённости, но можно применить и более строгое разделение с использованием интерфейса авторизации.[137] По сути, вы даёте разрешения на чтение/запись во внутренние темы только тем сервисам, которые за них отвечают. Это может быть реализовано с помощью простой проверки во время выполнения (англ. runtime validation) или же с обеспечением максимальной защиты через TLS или SASL.

Заключение

В этой главе мы рассмотрели несколько отдельных проблем, связанных с событийно-ориентированными системами. В первую очередь, мы обсудили изменение схемы данных — процесс, который является неотъемлемой частью современного мира. Зачастую этот вопрос решается путём изменения схемы в форматах, которые поддерживают обратную совместимость, таких как Avro или Protobuf. Однако это не всегда возможно, и иногда требуется идти и на «несовместимые» изменения. Один из вариантов внедрения такого изменения — параллельное использование двух схем, которое даёт сервисам окно времени для обновления.

Затем мы кратко рассмотрели проблему обработки нечитаемых сообщений, а также способы удаления данных. Для многих пользователей удаление данных не сопряжено ни с какими трудностями — данные просто

удаляются из журнала по истечении срока хранения. Однако для тех, кто хранит данные в течение более длительных периодов времени, этот вопрос представляет особый интерес.

ЧАСТЬ V

Создание потоковых сервисов с помощью Kafka

Реактивная система не занимается расчётами и функциями — она поддерживает отношения со своей средой на заданном уровне.

— Дэвид Харел и Амир Пнуэли

ГЛАВА 14

Kafka Streams и KSQL

Для построения событийно-ориентированных сервисов отлично подходит Kafka Streams API, в нём есть самый полный набор инструментов для управления распределёнными, асинхронными процессами. Kafka Streams API предназначен для выполнения потоковых вычислений. Мы рассмотрели простой пример таких вычислений в Главе 2, где шла обработка событий open/close в мобильном приложении. Затем, в Главе 6, мы поговорили об элементах состояния в потоковой обработке. Это привело нас к пониманию трёх типов сервисов: (1) событийно-ориентированных, (2) потоковых, (3) потоковых с сохранением состояния.

В этой главе мы более подробно рассмотрим этот уникальный инструмент для потоковой обработки с сохранением состояний, а также его мощный декларативный интерфейс — KSQL.

Разработка простого сервиса отправки почты с помощью Kafka Streams и KSQL

Kafka Streams является ключевым API-интерфейсом потоковой обработки для языков Java Virtual Machine (далее — JVM) — Java, Scala, Clojure и т. д. Он основан на DSL (предметно-ориентированный язык), который реализует декларативный интерфейс, через который потоки можно соединять, фильтровать, группировать или агрегировать средствами самого DSL. В этом языке также реализованы функциональные методы (`map`, `flatMap`, `transform`, `peek` и т. д.) для гибкой обработки каждого сообщения по отдельности. Важно отметить, что в создаваемых сервисах

можно совмещать оба подхода: с одной стороны, декларативный интерфейс даёт высокоуровневую абстракцию для SQL-подобных операций, с другой — на более низком уровне абстракции, функциональные методы дают возможность дорабатывать код в нужном вам направлении.

Но что, если ваша система работает не на JVM? В этом случае можно использовать KSQL. KSQL представляет собой простую, интерактивную SQL-подобную оболочку для Kafka Streams API. Она может работать автономно, например, используя шаблон «мотоколяска» (англ. sidecar),[138] и вызываться удалённо. Поскольку KSQL использует Kafka Streams API, он поддерживает те же декларативные функции. Кроме того, мы можем реализовать обработку данных под конкретную задачу двумя способами: либо (1) напрямую реализовать нужную пользовательскую функцию (англ. user-defined function, далее – UDF);[139] либо, что встречается чаще — (2) настроить вывод результатов запроса в тему Kafka, и используя библиотеку Kafka клиента в сервисе (он может быть написан на любом языке), получать эти темы и последовательно обрабатывать их сообщения, по одному за раз.

Какой бы подход вы ни выбрали, эти инструменты позволяют моделировать асинхронные, неблокирующие, независимые бизнес-операции. Давайте рассмотрим конкретный пример. Предположим, у нас есть сервис, который рассылает электронные письма «платиновым» покупателям (рис. 14.1). Эту задачу можно разделить на две части. Во-первых, нужно объединить поток заказов с таблицей покупателей и отфильтровать «платиновых» покупателей. Во-вторых, нужно написать код для создания и отправки самого электронного письма. Первую часть задачи можно решить через DSL, а вторую — через последовательную обработку сообщений:[140]

```
//Объединяем покупателей и заказы
orders.join(customers, Tuple::new)

    //Фильтруем подтверждённые заказы "платиновых" покупателей
    .filter((k, tuple) → tuple.customer.level().equals(PLATINUM)

&& tuple.order.state().equals(CONFIRMED))

    //Отравляем сообщение для каждой пары покупатель/заказ
    .peek((k, tuple) → emailer.sendMail(tuple));
```

Ту же самую операцию можно реализовать с помощью KSQL (рис. 14.2). Логика такая же: декларативное выражение разделяет поток событий, затем происходит обработка потока по одной записи за раз.

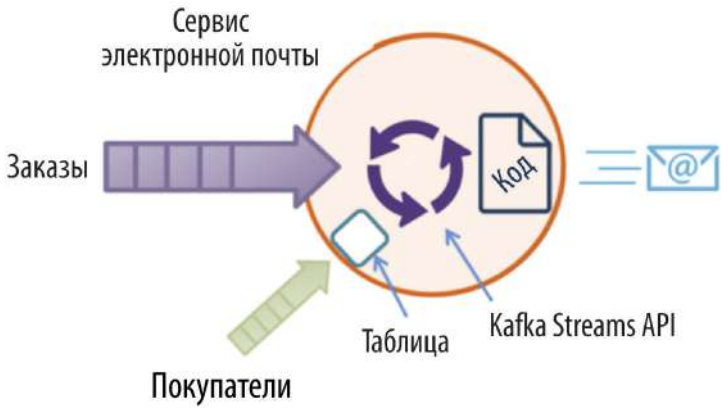


Рис. 14.1. Пример сервиса электронной почты, который объединяет заказы и покупателей, а затем отправляет письмо

```
//Создаем поток подтверждённых заказов от «платиновых» покупателей
ksql> CREATE STREAM platinum_emails AS SELECT * FROM orders
WHERE client_level == 'PLATINUM' AND state == 'CONFIRMED';
```

Затем мы реализуем сервис отправки сообщений в виде простого подписчика. Для этого используем API для Node.js (или любого другого поддерживаемого языка[141]) и KSQL с шаблоном «мотоколяска».

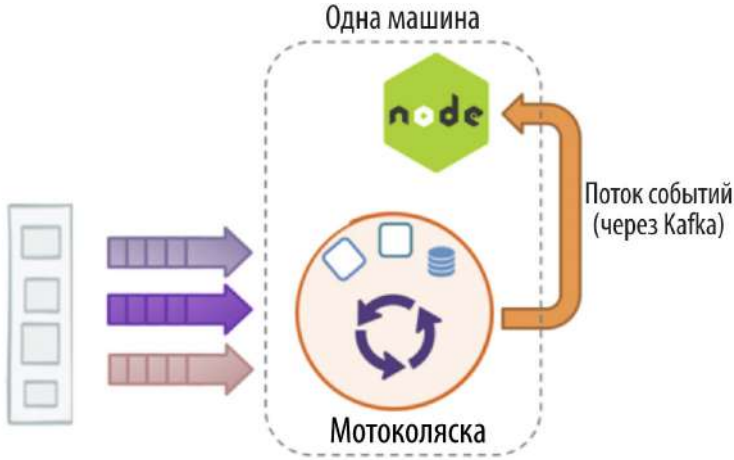


Рис. 14.2. Операция потоковой передачи выполняется через шаблон «мотоколяска»; обработка результирующего потока выполняется в клиенте Node.js

Оконные функции, соединения, таблицы и хранилище состояний

В Главе 6 мы говорили о том, что внутри Kafka Streams API можно хранить целые таблицы, что позволяет сервисам сохранять состояния. В этом разделе мы немного подробнее рассмотрим совместную работу потоков данных и таблиц, а также некоторых других ключевых элементов.

Давайте ещё раз вернемся к примеру с сервисом почты и рассмотрим процесс отправки письма с подтверждением оплаты нового заказа (рис. 14.3). Мы соединяем два потока событий (англ. stream-stream join). Как только сервис электронный почты обнаружит соответствие «заказ-платеж», будет запущен процесс отправки подтверждения покупателю. По сути, соединение работает так же, как логический оператор AND.

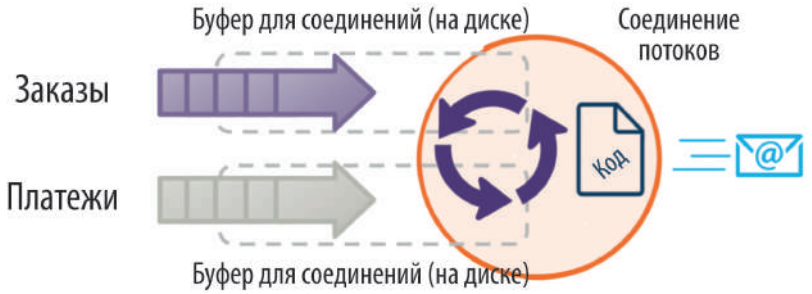


Рис. 14.3. Соединение потоков заказов и платежей

Потоки входящих событий буферизуются в течение определённого периода времени (он же — время хранения[142]). Чтобы не хранить все эти буферы в памяти, они направляются в хранилище состояний[143] на диске. Поэтому независимо от того, какое событие придёт позже, соответствующее ему событие другого потока можно быстро извлечь из буфера, чтобы операция объединения могла завершиться.

Kafka Streams также управляет целыми таблицами. Таблицы являются локальным объявлением целой темы (обычно уплотнённой). Они находятся в хранилище состояний и доступны по ключу. (Можно рассматривать их как потоки с бесконечным временем хранения.) В контексте сервисов такие таблицы зачастую используются для дополнения данных. Таким образом, для поиска адреса электронной почты покупателя можно использовать таблицу, загруженную из темы Покупатели в Kafka.

Такие таблицы хороши своим сходством с таблицами базы данных. Следовательно, когда мы объединяем поток заказов с таблицей покупателей, не нужно беспокоиться о времени хранения, оконных

функциях или других подобных сложностях. На рис. 14.4 изображено трёхстороннее соединение заказов, платежей и покупателей, где информация о покупателях представлена в виде таблицы.

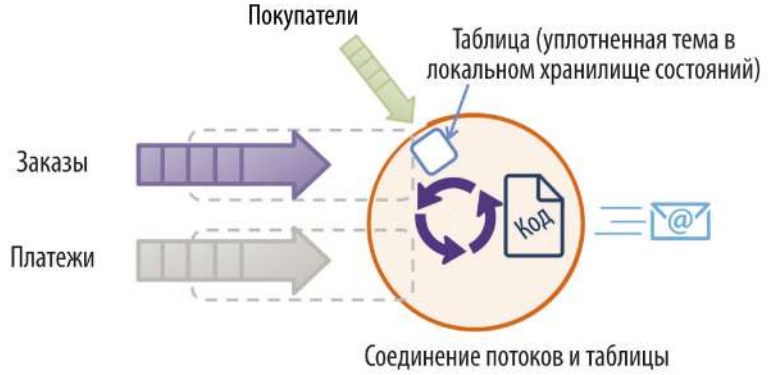


Рис. 14.4. Трёхстороннее соединение двух потоков и одной таблицы

На самом деле, в Kafka Streams есть два типа таблиц: KTable и Global KTable. Когда работает только один экземпляр сервиса, эти таблицы ведут себя одинаково. Однако при масштабировании сервиса и параллельной работе, скажем, четырёх его экземпляров, различия будут заметны. Дело в том, что таблицы типа Global KTable транслируются на экземпляры сервиса, и каждый из них получает полную копию всей таблицы. Таблицы типа KTable, в свою очередь, используют секционирование: набор данных разделяется и распределяется по всем экземплярам сервиса.

Эти типы таблиц по-разному выполняют операцию соединения. Поскольку в случае Global KTable полная её версия существует на каждом узле, её можно присоединять по любому удобному атрибуту — аналогично соединению по внешнему ключу в базе данных. KTable работает иначе. Поскольку такая таблица разделена по секциям, её можно присоединять к потокам только по первичному ключу. Таким образом, чтобы присоединить KTable или поток по атрибуту, отличному от первичного ключа, нужно произвести повторное секционирование. Об этом мы поговорим подробнее в разделе «Переназначение ключей для соединения» (Глава 15, стр. 160).

Итак, таблицы типа Global KTable хорошо работают в качестве таблиц поиска или соединений типа «звезда»,[144] но занимают больше места на диске, так как полностью транслируются на каждый экземпляр сервиса. Таблицы типа KTable позволяют масштабировать сервисы для больших наборов данных, однако в них может потребоваться переназначение ключей.[i]

i На самом деле, различие между этими двумя типами таблиц чуть более тонкое. См. Allow KTable bootstrap. *Apache*. - URL: https://issues.apache.org/jira/browse/KAFKA-4113

Хранилища состояний можно также использовать для хранения информации, как в обычных базах данных (рис. 14.5). Все записи, которые мы туда сохраняем, могут быть прочитаны позднее, например, после перезапуска сервиса. Таким образом, на нашем сервисе электронной почты можно развернуть интерфейс системного администратора, который собирает статистику отправленных писем. Эту статистику можно сохранять в хранилище состояний. Данные при этом будут не только сохраняться локально, но и записываться в соответствующий журнал изменений (под темой изменений) в Kafka, а значит, не могут быть потеряны, даже в случае сбоев (гарантия устойчивости Kafka).

Рис. 14.5. Сохранение состояний сервиса в хранилище состояний Kafka Streams API и тему Kafka

Заключение

В этой главе мы кратко познакомились с потоками данных, таблицами и хранилищами состояний — тремя важнейшими элементами потоковых приложений. Потоки данных являются бесконечными, и мы обрабатываем их последовательно, одну запись за другой. Таблицы представляют собой набор данных, который хранится локально. К ним можно присоединяться точно так же, как к таблицам базы данных. Хранилища состояний ведут себя как выделенные базы данных. Можно напрямую записывать туда и читать оттуда любую необходимую информацию. Это, конечно, только верхушка айсберга. В Kafka Streams и KSQL существуют гораздо больше элементов, некоторые из которых мы рассмотрим в Главе 15. В любом случае, все они основаны на этих базовых понятиях.

ГЛАВА 15

Построение потоковых сервисов

Экосистема подтверждения заказа

В предыдущих главах мы получили общее представление о работе Kafka Streams. Теперь давайте рассмотрим конкретные инструменты, необходимые для разработки небольшого потокового приложения. Для примера возьмём простой процесс обработки заказов, который проверяет и обрабатывает заказы по HTTP-запросам. Через REST API эти запросы попадают из синхронного мира в асинхронный, мир событий, а затем возвращаются обратно.[145]

Рассмотрим рис. 15.1 слева направо. REST API предоставляет методы для Отправки (далее — POST-запрос) и Получения (далее — GET-запрос) информации о заказах. После получения POST-запроса сервис заказов Kafka создаёт событие «Заказ создан». Три сервиса подтверждения (сервис проверки на мошенничество, сервис управления запасами, сервис детализации заказов) подписываются на эти события. Сервисы работают параллельно и выдают результат проверки в формате PASS или FAIL. Результаты этих проверок записываются в отдельную тему «Подтверждения заказов», чтобы не нарушать принцип единственного писателя — в тему «Заказы» пишет только сервис заказов.[i] Совместный результат трёх проверок передаётся сервису заказов, который затем переводит заказ в статус «Подтверждён» или «Не подтверждён». Проверенные заказы накапливаются в представлении заказов, оттуда же можно запрашивать исторические данные — реализацию шаблона проектирования CQRS (см.

i В данном случае мы решили записывать результаты проверок в отдельную тему «Подтверждения заказов», однако можно было бы добавлять их напрямую в тему «Заказы», реализуя принцип писателя для смены состояния (Глава 11).

«Командно-запросная изоляция» в Главе 7 на стр. 67). Сервис электронной почты отправляет письмо с подтверждением.

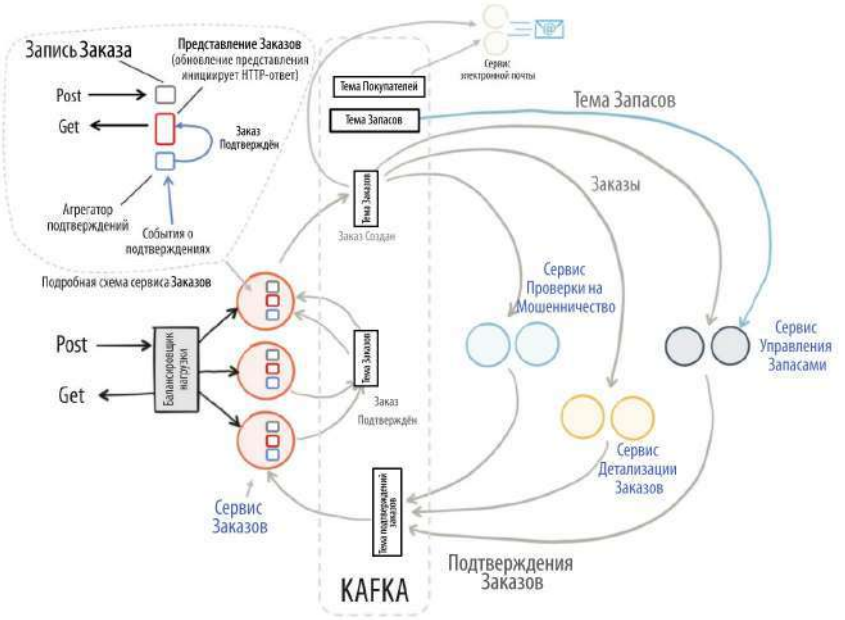


Рис. 15.1. Система обработки заказов, реализованная как набор потоковых сервисов

Сервис управления запасами проверяет заказ, а также резервирует необходимое количество товара на складе. Это даёт нам интересный сценарий, поскольку необходимо атомарно связать операции чтения и записи. Мы подробно рассмотрим эту проблему далее в этой главе.

Объединение-фильтрация-обработка

Большинство потоковых систем работает по схеме «соединение-фильтрация-обработка» (англ. join-filter-process), где сначала потоки проходят предварительную подготовку, а затем их последовательно обрабатывают, событие за событием. Этот процесс включает три шага:

1. Соединение. Для соединения множества потоков и таблиц, полученных от других сервисов, используется DSL.

2. Фильтрация. Оставляем только нужные значения, а всё остальное отбрасываем. На данном этапе также часто используется агрегирование.

3. Обработка. Функция принимает объединённые, отфильтрованные данные и применяет к ним бизнес-логику. Результат обработки передаётся в отдельный поток.

По такой схеме работают многие сервисы, но наиболее наглядным примером является сервис электронной почты. Он объединяет заказы, платежи и покупателей, а затем передаёт эти данные функции, которая отвечает за отправку сообщений. Обработку данных по такому шаблону можно реализовать как с помощью Kafka Streams, так и через KSQL.

Представление на основе событий в Kafka Streams

Чтобы пользователи могли отправить HTTP GET-запрос и получить данные о заказах (в т. ч. исторические), сервис заказов создаёт представление источника событий (См. «Представление источника событий» в Главе 7). Для этого заказы сохраняются в группы хранилищ состояний, распределённые по трём экземплярам представления заказов. Нагрузка и объём памяти также распределяются по всем экземплярам.

Рис. 15.2. Крупный план сервиса заказов, представленного на рис. 15.1. Сервис заказов создаёт материализованное представление, которое может быть получено через HTTP GET-запрос. Это представление — реализация «запросной» части шаблона CQRS — распределено по всем трём экземплярам сервиса заказов

Поскольку данные разделены на секции, их можно горизонтально масштабировать (Kafka Streams поддерживает динамическую перебалансировку под нагрузкой). Однако, это также означает, что GET-запросы должны направляться на соответствующий узел, тот, на котором, находится секция, соответствующая указанному в запросе ключу. Поиск соответствующего узла происходит автоматически через интерактивные запросы[146] Kafka Streams.[ii]

ii Также распространённой практикой является реализация представлений на основе событий с помощью Kafka Connect и любой базы данных по вашему выбору (см. «Делаем запросы к оптимизированному для чтения представлению в БД» в Главе 7 на стр. 76). Используйте этот метод, если вам нужны более разнообразные запросы или больший объем памяти.

На самом деле, этот процесс состоит из двух частей. Во-первых, нужно написать запрос, который определяет, какие данные должны уходить в представление. В данном случае нужно секционировать поток заказов по первичному ключу (старые заказы будут перезаписаны новыми) и записать результат в хранилище состояний, к которому можно сделать запрос. Можно реализовать этот шаг с помощью Kafka Streams DSL:

```
builder.stream(ORDERS.name(), serializer)
    .groupByKey(groupSerializer)
    .reduce((agg, newVal) -> newVal, getStateStore())
```

Во-вторых, необходимо, чтобы HTTP-сервер мог обращаться к хранилищу(-ам) состояний. Это несложно реализовать, однако при наличии нескольких экземпляров сервиса нужно направлять запросы к тому экземпляру и секции, которые соответствуют указанному в запросе ключу. Kafka Streams координирует запросы автоматически с помощью сервиса метаданных.

Нарушение принципа CQRS через блокировку чтения

Сервис заказов блокирует HTTP GET-запросы, чтобы клиенты имели возможность считывать свои собственные записи. Этот приём используется для обхода асинхронной природы шаблона CQRS. Так, например, если клиент выполняет запись, а затем сразу же чтение, событие может не успеть попасть в сервис представления. Результатом будет сообщение об ошибке или некорректное значение.

В качестве решения этой проблемы можно заблокировать GET-запросы до тех пор, пока нужное событие не окажется в сервисе представления (или пока не истечёт заданное время ожидания). Таким образом, мы нарушаем асинхронную работу CQRS, и на стороне клиента запись и чтение происходят последовательно. По сути, это реализация механизма «длинных запросов» (англ. long polling). В примере кода, приведённом выше, сервис заказов реализует этот механизм с помощью неблокирующего ввода-вывода (I/O).

Масштабирование параллельных операций в потоковых системах

Сервис управления запасами представляет особый интерес, поскольку для его корректной, бесперебойной работы требуется реализация нескольких

специальных механизмов. Сервис выполняет простую операцию: когда пользователь покупает iPad, сервис (1) проверяет, что на складе осталось достаточное количество iPad-ов, (2) физически резервирует необходимое их количество, чтобы эти единицы товара не были доступны никакому другому процессу (рис. 15.3). Однако все чуть сложнее, чем кажется, поскольку данная операция подразумевает атомарное чтение и запись в разные темы. В частности, необходимо:

1. Проверить, что на складе достаточно iPad-ов (количество iPad-ов на складе за вычетом тех, что уже зарезервированы).
2. Обновить таблицу «Зарезервированное количество», чтобы этот iPad не достался какому-то другому покупателю.
3. Отправить сообщение о подтверждении заказа.

Рис. 15.3. Сервис управления запасами подтверждает заказы, проверяя достаточное наличие товаров на складе, а затем резервирует необходимое их количество в хранилище состояний (данные из хранилища дублируются в тему Kafka). Все операции инкапсулируются в транзакцию

Сервис управления запасами будет надёжно работать только в том случае, если:

- Будет включена функциональность транзакций Kafka;
- Перед началом операции данные будут разделены на секции по `ProductId`.

Первый пункт является достаточно очевидным: если не инкапсулировать все шаги в транзакцию, то неизвестно, что в итоге получится. Чтобы лучше понять смысл второго пункта, достаточно представить, что мы горизонтально масштабируем эту операцию — распределяем её выполнение на несколько разных потоков (англ. threads) или машин.

В системах потоковой обработки с сохранением состояния, таких как Kaf-

ka Streams, реализован новый механизм для эффективного управления такими параллельными процессами. В данном случае у нас есть одна критическая секция:

1. Прочитать количество доступных iPad-ов на складе.
2. Зарезервировать количество iPad-ов, указанное в заказе.

Для начала давайте посмотрим, как работала бы в данном случае традиционная потоковая система, т. е. система без сохранения состояния (рис. 15.4). Если бы мы масштабировали операцию подтверждения и запустили два параллельных процесса, необходимо было бы создать критическую секцию внутри транзакции (общей) базы данных. Таким образом, в любой момент времени только один из экземпляров сервиса мог бы получить доступ к базе данных.

Рис. 15.4. Два экземпляра сервиса управляют параллельными операциями через общую базу данных

Системы потоковой обработки с сохранением состояния, такие как Kafka Streams, избегают использования удаленных транзакций и координацию процессов между собой. Для этого используется секционирование — разделение на несколько потоков или процессов на основе выбранного бизнес-логикой ключа (мы обсуждали «Секции и секционирование» в Главе 4). Именно этот шаг является ключевым (простите за каламбур) элементом для горизонтального масштабирования таких систем.

Секционирование в Kafka Streams работает следующим образом: сообщения перенаправляются так, чтобы вся информация о состояниях, необходимая для конкретного вычисления, находилась в одном месте.[iii] В рамках данного подхода операции выполняются параллельно, благодаря чему потоковые системы и достигают такой высокой скорости обработки сообщений (например, в рамках сценария, который мы обсуждали в Главе 2). Однако этот подход работает только при наличии логического ключа,

iii Кстати, одной из приятных особенностей секционирования является то, что управляет им Kafka, а не Kafka Streams. Протокол Kafka Consumer Group Protocol позволяет любой группе подписчиков контролировать распределение секций в рамках своей группы. См. Consumers. *Apache Kafka*. - URL: https://kafka.apache.org/0110/documentation/#intro_consumers

который чётко разделяет все операции: как чтение состояния, так и его запись.

Таким образом, разделение (т. е. секционирование) данных по ProductId гарантирует, что все операции для одного ProductId будут последовательно выполняться одним и тем же процессом. Это означает, что все iPad будут обрабатываться одним процессом, и все iWatch также будут обрабатываться одним (вероятно, отдельным) процессом, при этом процессы не будут взаимодействовать друг с другом при выполнении критической секции (рис. 15.5). В результате мы получаем атомарную (благодаря гарантиям Kafka), горизонтально масштабируемую операцию, которая не требует ресурсозатратного взаимодействия между сетями. (Это напоминает реализацию фазы Map в системах на основе MapReduce).

Рис. 15.5. Сервисы, использующие Kafka Streams API, распределяют потоки событий и хранилища состояний по разным экземплярам и сервисам. В результате все данные, необходимые для выполнения критической секции, располагаются локально — к ним обращается только один процесс

Необходимо, чтобы сервис управления запасами перераспределил заказы по ProductId. Для этого используется операция «переназначения ключей» (англ. rekey): заказы записываются в новую промежуточную тему Kafka, на этот раз с привязкой к ProductId, а затем передаются обратно в сервис управления запасами. Реализовать это очень просто:

```
orders.selectKey((id, order) -> order.getProduct())//rekey by ProductId
```

Вторая часть критической секции представляет собой смену состояния: товары нужно зарезервировать. Сервис управления запасами выполняет данный шаг через хранилище состояний (локальная хэш-таблица на жёстком диске, которая дублируется в тему Kafka) в Kafka Streams. Таким образом, каждому процессу будет доступно хранилище состояний, где находится информация о зарезервированных единицах товара той или иной категории. Вы можете работать с хранилищами состояний

почти так же, как с хэш-таблицами или хранилищами типа «ключ-значение». Преимуществом использования хранилищ состояний является устойчивость данных — при перезапуске процесса они восстанавливаются. Можно создать хранилище состояний с помощью одной строки кода:

```
KeyValueStore<Product, Long> store = context.getStateStore
(RESERVED);
```

Затем, мы используем это хранилище почти так же, как обычную хэш-таблицу:

```
//Получить текущее количество зарезервированных единиц данного товара

Long reserved = store.get(order.getProduct());

//Добавить количество единиц необходимое для данного заказа и отправить обратно

store.put(order.getProduct(), reserved + order.getQuantity())
```

На запись в хранилище состояний также распространяются гарантии транзакций Kafka, о которых мы говорили в Главе 12.

Переназначение ключей для соединения

Тот подход, что мы использовали в предыдущем пункте для секционирования записей, можно применить и для секционирования чтений (например, для выполнения операции соединения). Предположим, нужно присоединить поток заказов (ключ — OrderId) к таблице запасов (ключ — ProductId), как показано на рис. 15.3. Для операции соединения придётся использовать ProductId.

В реляционных базах данных такой атрибут называется внешний ключ. Таким образом, мы устанавливаем соответствие между WarehouseInventory.ProductId (первичный ключ таблицы запасов) и Order.ProductId (внешний ключ потока заказов). Для этого нужно перераспределить заказы по узлам так, чтобы они обрабатывались тем процессом, который владеет информацией по складским запасам товара с соответствующим ключом.

Как упоминалось ранее, этот этап перераспределения данных называется «переназначение ключей», а данные, распределённые таким образом, называются «совместно секционированными (англ. co-partitioning)».[147] Как только ключи были переназначены, можно выполнять соединение. Доступ к сети при этом не требуется. Например, на рис. 15.6 информация о складских запасах для productId=5 расположена в том же экземпляре сервиса, где и заказы для productId=5.

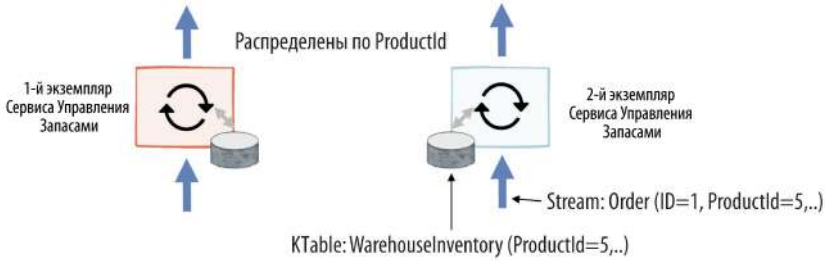


Рис. 15.6. Чтобы соединить заказы и складские запасы по ProductId, нужно перераспределить заказы по ProductId. Это гарантирует, что все заказы, соответствующие определённому товару, будут находиться на одном и том же экземпляре сервиса

Перераспределение данных и поэтапное исполнение

Настоящие системы обычно гораздо более сложны и многогранны. В одну минуту мы выполняем соединение, а в следующую — агрегируем данные по покупателям или выводим их материализованное представление, причём для каждой операции требуется по-разному распределять данные. Все эти различные операции связываются в единый последовательный процесс. Наглядный пример — сервис управления запасами. Он использует операцию переназначения ключей, чтобы распределить данные по ProductId. Сразу после завершения этой операции данные необходимо перераспределить обратно по OrderId, чтобы можно было их вывести в представление заказов (рис. 15.7). (Это представление работает деструктивно — старые версии заказа перезаписываются более новыми. Поэтому, чтобы избежать потери данных, в качестве ключа потока заказов должен использоваться OrderId.)

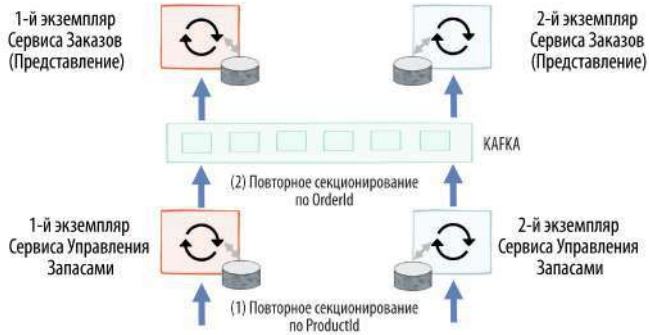


Рис. 15.7. Два этапа, которые требуют объединения данных по разным ключам, связаны в единый последовательный процесс через операцию переназначения ключей: сначала используется ProductId, затем — OrderId

Однако у этого подхода есть свои ограничения. Для того, чтобы гарантировать упорядочивание сообщений, ключи, используемые для секционирования потоков событий, должны оставаться неизменными. В нашем случае это означает, что ProductId и OrderId для каждого заказа должны оставаться фиксированными во всех сообщениях, связанных с определённым заказом. Обычно этим можно довольно легко управлять на уровне бизнес-логики (например, создать правило, согласно которому при изменении пользователем товаров в корзине должен быть сгенерирован новый заказ).

Ожидание N событий

Ещё один относительно распространённый в бизнес-системах сценарий — ожидание N событий. Если каждое событие находится в отдельной теме, то этот механизм можно легко реализовать с помощью трёхстороннего соединения. Однако, если все события записываются в одну тему, всё несколько сложнее.

В рассмотренном ранее примере (рис. 15.1) сервис заказов ожидает результатов проверки от каждого из трёх сервисов подтверждения. Все три сервиса пишут в одну и ту же тему. Проверка считается пройденной успешно только, если все три сервиса вернули результат PASS. Предположим, мы считаем сообщения, в которых содержится определённый ключ. Тогда получаем следующее решение:

1. Произвести секционирование по ключу.
2. Посчитать количество вхождений каждого ключа (с помощью оконного агрегирования).
3. Отфильтровать полученные данные по необходимому числу вхождений.

Размышления на тему архитектуры

У любой распределённой системы есть своя базовая цена. Это должно быть очевидно. Описанное здесь решение обеспечивает хорошую масштабируемость и отказоустойчивость, но всегда будет более сложным в реализации и использовании, чем простое однопроцессное приложение, разработанное для выполнения той же логики. При разработке системы нужно находить необходимый баланс между нефункциональными свойствами (надёжность, производительность и т. д.) и простотой. При этом стоит подчеркнуть, что настоящие системы по сравнению с нашими примерами всегда будут гораздо более сложными, с большим количеством элементов. Поэтому первоначальные затраты на реализацию такой модульной, расширяемой архитектуры, скорее всего, окупятся в будущем.

Более комплексная потоковая экосистема

В этом последнем пункте мы кратко рассмотрим более крупную экосистему сервисов (рис. 15.8). Она объединяет часть ключевых элементов, которые мы обсудили в рамках этой книги. Мы также поговорим про роль каждого из этих сервисов и шаблоны их реализации:

Запись в тему Корзины / Представление Корзины
: Эти сервисы представляют собой реализацию принципа CQRS (см. «Командно-запросная изоляция» в Главе 7 на стр. 67). Когда пользователь добавляет новую единицу товара в корзину, HTTP-запрос перенаправляется в тему Корзины. Для этого используется прокси-сервер Confluent REST Proxy (поставляется с дистрибутивом Kafka). Представление Корзины — представление на основе событий, реализованное в Kafka Streams. Аналогично примеру с сервисом заказов, рассмотренному ранее в этой главе, к хранилищам состояний представление Корзины может обращаться через REST API (также можно использовать Kafka Connect и базу данных). Представление Корзины является объединением тем Пользователя и Корзины, где большая часть информации была отфильтрована. Остался только необходимый минимум: `userId` → `List[product]`. Это минимизирует объём занимаемой памяти.

Представление Фильтров Каталога
: Это еще одно представление на основе событий, которое требует расширенной функциональности для разбиения результатов запроса на страницы, поэтому для реализации этого сервиса используются Kafka Connect и Cassandra.

Поиск по Каталогу
: Это третье представление на основе событий; использует платформу Solr для реализации полнотекстового поиска.

Сервис Заказов
: Заказы подтверждаются и сохраняются в Kafka. Этот процесс может быть реализован либо в виде отдельного сервиса, либо в виде небольшой экосистемы, наподобие той, что мы рассмотрели ранее в этой главе.

Сервис Каталога
: Старая кодовая база, которая управляет изменениями, внесёнными в продуктовый каталог через UI. У этого сервиса сравнительно меньше пользователей, и есть существующая кодовая база. Коннектор с поддержкой отслеживания измененных данных (CDC) передаёт

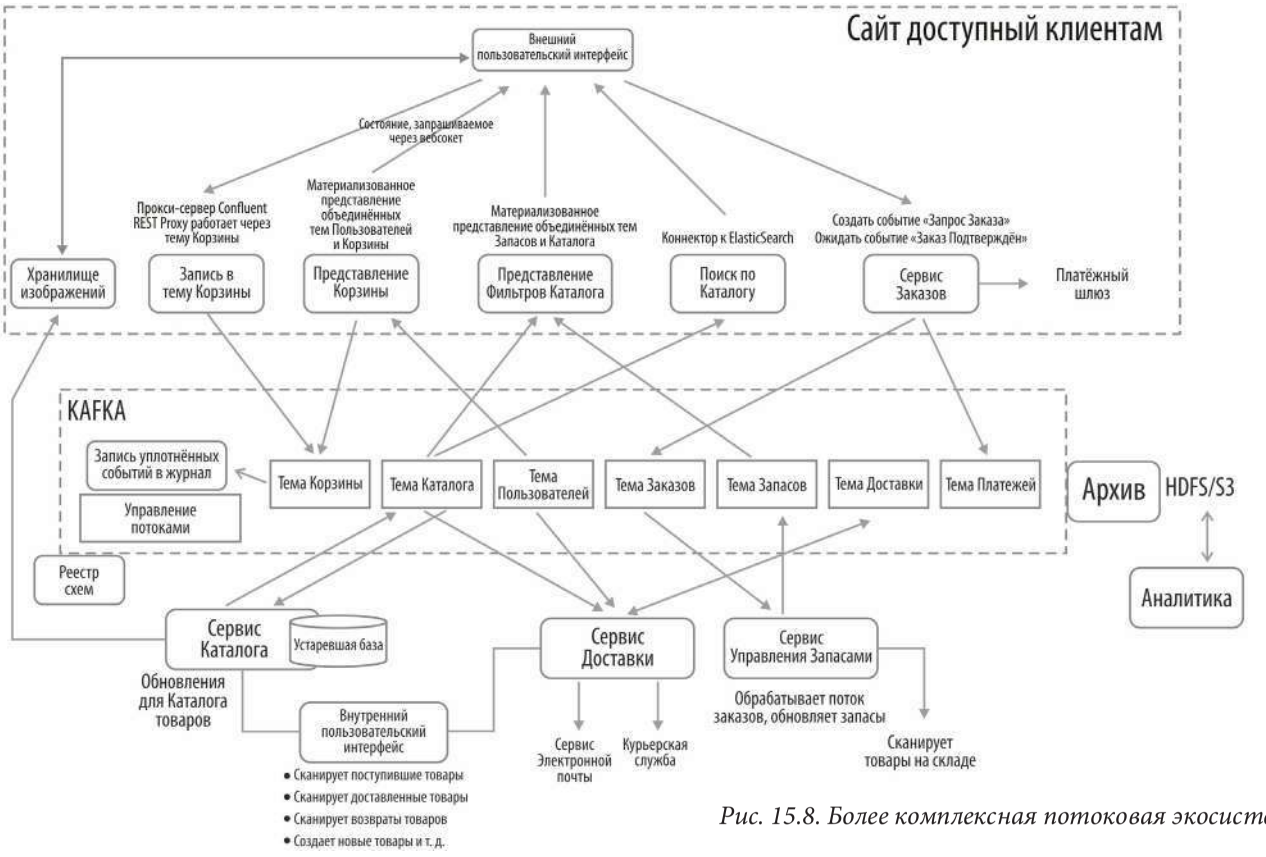


Рис. 15.8. Более комплексная потоковая экосистема

события из старой базы данных в Postgres в Kafka. Функция SMT (англ. single message transforms) меняет формат сообщений перед их публикацией. Изображения хранятся в распределённой файловой системе для упрощённого доступа к ним через веб-уровень.

Сервис Доставки
: Потоковый сервис, использующий Kafka Streams API. Этот сервис реагирует на появление новых заказов и обновляет тему Доставка по мере поступления уведомлений от компании, которая осуществляет доставку товаров.

Сервис Управления Запасами
: Ещё один потоковый сервис, использующий Kafka Streams API. Этот сервис обновляет информацию о запасах по мере того, как товары поступают на склад и покидают его.

Архив
: Все события сохраняются в файловой системе HDFS, включая два фиксированных снимка данных (в моменты времени T-1 и T-10)[148] для целей восстановления. Сервис использует Kafka Connect и HDFS-коннектор.

Управление потоками
: Набор сервисов обработки потоков при необходимости управляет созданием тем актуальный /версионированный (см. «актуальный-версионированный» шаблон в разделе «Долговременное хранение данных» на стр. 28 в Главе 4). Эти сервисы также управляют вспомогательными темами, которые используются при необходимости развернуть «несовместимое» изменение схемы. (См. «Управление изменениями в схеме данных и нарушение обратной совместимости» в Главе 13 на стр. 136).

Реестр схем
: Реестр схем — репозиторий схем от Confluent — обеспечивает проверку схем и их совместимость во время выполнения.

Заключение

При создании потоковых систем используются разные виды сервисов. Некоторые не сохраняют состояние — это простые функции, которые принимают данные, выполняют бизнес-операции и выдают результат. Некоторые сервисы сохраняют состояние, но работают только на

чтение, например, представления на основе событий. Другие типы сервисов могут как читать, так и записывать состояние: либо полностью внутри экосистемы Kafka (и, следовательно, с соблюдением гарантий на транзакции), либо через использование других сервисов или баз данных. Одним из наиболее привлекательных свойств API-интерфейсов потоковой обработки с сохранением состояния является то, что нам доступны все эти параметры. Всегда можно обменять простоту операций, выполняемых без сохранения состояния, на расширенные возможности обработки данных, характерные для сервисов с сохранением состояния.

Однако у этого подхода есть, конечно, и свои недостатки. Несмотря на то, что локальные копии баз данных, контрольные точки и уплотнённые темы снижают риски, связанные с передачей данных в код, всегда возможна реализация худшего сценария, когда необходимо перестраивать находящийся в сервисе набор данных. Нужно относиться к этому как к неотъемлемой части любой архитектуры.

Работа с потоковыми системами, помимо прочего, требует другого образа мышления. По сравнению с традиционными интерфейсами, работающими на принципах императивного программирования, потоковая модель работает с асинхронными процессами и придерживается более функционального и ориентированного на данные стиля разработки. На мой, авторский, взгляд, потоковые системы являются выгодным вложением ресурсов.

В этой главе мы рассмотрели очень простую систему обработки заказов, состоящую из набора небольших потоковых микросервисов, которые работают по принципу событийного сотрудничества (мы говорили о нем в Главе 5). В заключение мы разобрали вариант с более масштабной архитектурой, где используется более широкий набор шаблонов, о которых мы говорили в этой книге.

Об авторе

Бен Стопфорд — технический специалист из отдела CTO в корпорации Confluent (компания, стоящая за Apache Kafka), где он работает над широкой группой проектов, начиная с внедрения последних версий протокола репликации Kafka и заканчивая разработкой стратегии для потоковых приложений. До прихода в Confluent, Бен руководил проектированием и построением платформы данных для крупной финансовой организации, а также работал над целым рядом ранних сервис-ориентированных систем как в финансовой сфере, так и в ThoughtWorks. Бен регулярно выступает на конференциях, ведёт блог. Он внимательно наблюдает за технологиями и уверен, что мы вступаем в интересную эпоху, когда специалисты по обработке данных, разработчики и жизненный цикл организаций всё больше влияют друг на друга.

Список литературы

1. Newman S. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media. 2015. - URL: http://shop.oreilly.com/product/0636920033158.do.

2. Parnas D. L. On the Criteria to Be Used in Decomposing Systems into Modules. *Carnegie Mellon University Research Showcase*. 1971. - URL: https://prl.ccs.neu.edu/img/p-tr-1971.pdf.

3. Chitchcock. *Stevey's Google Platforms Rant*. 2011. - URL: https://gist.github.com/chitchcock/1281611.

4. Helland P. *Immutability Changes Everything*. - URL: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper16.pdf.

5. См. Chitchcock.

6. Stafford P. What is Mifid II and how will it affect EU's financial industry? *Financial Times*. September 15, 2017. - URL: https://www.ft.com/content/ae935520-96ff-11e7-b83c-9588e51488a0.

7. ESB. *ThoughtWorks*. - URL: https://www.thoughtworks.com/radar/tools/esb.

8. Fowler M. IntegrationDatabase. *MartinFowler.Com*. May 25, 2004. - URL: https://martinfowler.com/bliki/IntegrationDatabase.html.

9. Helland P. *Data on the Outside versus Data on the Inside*. - URL: http://cidrdb.org/cidr2005/papers/P12.pdf.

10. Kreps J. Questioning the Lambda Architecture. *O'Reilly*. July 2, 2014. - URL: https://www.oreilly.com/ideas/questioning-the-lambda-architecture.

11. Hickey R. Deconstructing the Database. *InfoQ*. - URL: https://www.youtube.com/watch?v=-Cym4TZwTCNU.

12. Event streaming as the source of truth. *ThoughtWorks* - URL: https://www.thoughtworks.com/radar/techniques/event-streaming-as-the-source-of-truth.

13. Kleppmann M. Turning the database inside-out with Apache Samza. *Confluent*. March 1, 2015. - URL: https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/.

14. Streaming SQL for Apache Kafka. *Confluent*. - URL: https://www.confluent.io/product/ksql/.

15. Interactive Queries. *Confluent*. - URL: https://docs.confluent.io/4.0.0/streams/developer-guide/interactive-queries.html.

16. См. ESB.

17. Recreating ESB antipatterns with Kafka. *ThoughtWorks*. - URL: https://www.thoughtworks.com/radar/techniques/recreating-esb-antipatterns-with-kafka.

18. Event streaming as the source of truth. *ThoughtWorks*. - URL: https://www.thoughtworks.com/radar/techniques/event-streaming-as-the-source-of-truth.

19. Kreps J. Putting Apache Kafka To Use: A Practical Guide to Building an Event Streaming Platform

(Part 1). *Confluent*. February 25, 2015. - URL: https://www.confluent.io/blog/event-streaming-platform-1.

20. Discover and share Connectors and more. *Confluent*. - URL: https://www.confluent.io/hub/.

21. Java Message Service. *Wikipedia*. - URL: https://ru.wikipedia.org/wiki/Java_Message_Service.

22. AMQP. *Wikipedia*. - URL: https://ru.wikipedia.org/wiki/AMQP.

23. Kreps J. The Log: What every software engineer should know about real-time data's unifying abstraction. *LinkedIn*. December 16, 2013. - URL: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying.

24. Lampkin V. Large messages depths on your WebSphere MQ queues could cause performance issues. *IBM Community*. May 4, 2013. - URL: https://www.ibm.com/developerworks/community/blogs/aimsupport/entry/large_messages_depths_on_mq_queues?lang=en.

25. Transaction log. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Transaction_log.

26. См. Lampkin V.

27. См. Transaction log.

28. Fowler M. Event Sourcing. *MartinFowler.Com*. December 12, 2005. - URL: https://martinfowler.com/eaaDev/EventSourcing.html.

29. Summer B. Kafkapocalypse: Monitoring Kafka Without Losing Your Mind. *New Relic*. December 12, 2017. - URL: https://blog.newrelic.com/engineering/new-relic-kafkapocalypse/.

30. Kafka Inside Keystone Pipeline. *Netflix Technology Blog*. April 27, 2016. - URL: https://medium.com/netflix-techblog/kafka-inside-keystone-pipeline-dd5aeabaf6bb.

31. Enforcing Client Quotas. *Confluent*. - URL: https://docs.confluent.io/current/kafka/post-deployment.html#quotas.

32. Configuration. *Confluent*. - URL: https://docs.confluent.io/current/clients/producer.html#configuration.

33. Stopford B. Log Structured Merge Trees. *BenStopford*. - URL: http://www.benstopford.com/2015/02/14/log-structured-merge-trees/.

34. Kreps J. It's Okay To Store Data In Apache Kafka. *Confluent*. September 15, 2017. - URL: https://www.confluent.io/blog/okay-store-data-apache-kafka/.

35. Juma I. Apache Kafka Security 101. *Confluent*. February 1, 2016. - URL: https://www.confluent.io/blog/apache-kafka-security-authorization-authentication-encryption/.

36. Treynor B. et al. The Calculus of Service Availability. *Web Services*. May 17, 2017. - URL: https://queue.acm.org/detail.cfm?id=3096459.

37. Kreps J. Putting Apache Kafka To Use: A Practical Guide to Building an Event Streaming Platform (Part 1). *Confluent*. February 25, 2015. - URL: https://www.confluent.io/blog/event-streaming-platform-1.

38. Process Manager. *Enterprise Integration Patterns*. - URL: https://www.enterpriseintegrationpatterns.com/patterns/messaging/ProcessManager.html.

39. Separation of concerns. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Separation_of_concerns.

40. Stevens W. P. et al. Structured design. *IBM Systems Journal*. - URL: https://ieeexplore.ieee.org/document/5388187.

41. Leymann - Keynote ESOCC 2016. - URL: https://web.archive.org/web/20170510160254/http://esocc2016.eu/wp-content/uploads/2016/04/Leymann-Keynote-ESOCC-2016.pdf.

42. Connascence. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Connascence.
См. также Page-Jones M. Comparing techniques by means of encapsulation and connascence. *Communications Of The Acm*. - URL: http://wiki.cfcl.com/pub/Projects/Connascence/Resources/p147-page-jones.pdf.

43. Ford N. et al. *Building Evolutionary Architectures*. O'Reilly Media. 2017. - URL: http://shop.oreilly.com/product/0636920080237.do.

44. Fowler M. What do you mean by "Event-Driven"? *MartinFowler.com*. February 7, 2017. - URL: https://martinfowler.com/articles/201701-event-driven.html.

45. Data integration. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Data_integration.

46. Bonér J. *Reactive Microservices Architecture: Design Principles for Distributed Systems*. O'Reilly Media. - URL: https://www.lightbend.com/blog/reactive-microservices-architecture-free-oreilly-report-by-lightbend-cto-jonas-boner.

47. См. Process Manager. См. также D'Amore J. Scaling Microservices with an Event Stream. ThoughtWorks. May 5, 2015. - URL: https://www.thoughtworks.com/insights/blog/scaling-microservices-event-stream.

48. Domain-driven design. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Domain-driven_design.

49. См. Helland P. *Data on the Outside versus Data on the Inside.*

50. GRPC. - URL: https://grpc.io/.

51. Finagle. *Twitter*. - URL: https://twitter.github.io/finagle/.

52. Data-driven programming. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Data-driven_programming.

53. Kleppmann M. Apache Kafka, Samza, and the Unix Philosophy of Distributed Data. *Confluent*. August 1, 2015. - URL: https://www.confluent.io/blog/apache-kafka-samza-and-the-unix-philosophy-of-distributed-data/.

54. Calderwood B. Toward a Functional Programming Analogy for Microservices. *Confluent*. November 29, 2017. - URL: https://www.confluent.io/blog/toward-functional-programming-analogy-microservices/.

55. Purely functional programming. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Purely_functional_programming.

56. Functional Core, Imperative Shell. *DestroyAllSoftware*. - URL: https://www.destroyallsoftware.com/screencasts/catalog/functional-core-imperative-shell.

57. Parra R. Functional Core, Business events in a world of independently deployable services. *Medium*. January 6, 2018. - URL: https://medium.com/homeaway-tech-blog/business-events-in-a-world-of-independently-deployable-services-144daf6caa1a.

58. Num.standby.replicas. *Apache Kafka*. - URL: https://kafka.apache.org/10/documentation/streams/developer-guide/config-streams.html#num-standby-replicas.

59. State restoration during workload rebalance. *Apache Kafka*. - URL: https://kafka.apache.org/10/documentation/streams/developer-guide/running-app.html#state-restoration-during-workload-rebalance.

60. Log Compaction. *Apache Kafka*. - URL: https://kafka.apache.org/documentation.html#compaction.

61. Kafka Streams Application Reset Tool. *Confluence*. - URL: https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Application+Reset+Tool. См. также Sax M. J. Data Reprocessing with the Streams API in Kafka: Resetting a Streams Application. *Confluent*. August 15, 2016. - URL: https://www.confluent.io/blog/data-reprocessing-with-kafka-streams-resetting-a-streams-application/.

62. Fowler M. CQRS. *MartinFowler.com*. July 14, 2011. - URL: https://martinfowler.com/bliki/CQRS.html.

63. См. Fowler M. Event Sourcing и Fowler M. What do you mean by “Event-Driven”?

64. Create, read, update and delete. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Create,_read,_update_and_delete.

65. Staging (data). *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Staging_(data).

66. Row and statement level triggers. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Database_trigger#Row_and_statement_level_triggers.

67. Bitemporal Modeling. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Bitemporal_Modeling.

68. Write-ahead logging. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Write-ahead_logging.

69. Stonebraker M. and Çetintemel U. “One Size Fits All”: An Idea Whose Time Has Come and Gone. - URL: http://cs.brown.edu/~ugur/fits_all.pdf.

70. См. Domain-driven design. *Wikipedia*.

71. Discover and share Connectors and more. *Confluent*. - URL: https://www.confluent.io/hub/.

72. Change data capture. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Change_data_capture.

73. См. Discover and share Connectors and more. *Confluent*.

74. Feathers M. Testing Effectively With Legacy Code. *Working Effectively with Legacy Code*. January 21, 2005. - URL:http://www.informit.com/articles/article.aspx?p=359417&seqNum=3.

75. Transforms. *Confluent*. - URL: https://docs.confluent.io/3.3.0/connect/concepts.html#transforms.

76. Data Platforms Landscape Map. February 2014. - URL: https://blogs.the451group.com/information_management/files/2014/03/data_map.gif.

77. Fowler M. MemoryImage. *MartinFowler.com*. August 31, 2011. - https://martinfowler.com/bliki/MemoryImage.html.

78. Spolsky J. Things You Should Never Do, Part I. *Joel On Software*. April 6, 2000. - URL: https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/.

79. Big ball of mud. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Big_ball_of_mud.

80. DeMarco T. *Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency*. Broadway Books. 2001.

81. Divide and rule. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Divide_and_rule.

82. Inverse Conway Maneuver. *ThoughtWorks*. - URL: https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver.

83. См. Newman S. *Building Microservices*.

84. Tornhill A. Software (r)Evolution Part 3: Time - The Hidden Dimension of Software Design. *Empear*. October 19, 2016. - URL: https://www.empear.com/blog/software-revolution-part3/.

85. Master data management. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Master_data_management.

86. См. Helland P. *Data on the Outside versus Data on the Inside*.

87. См. Ford N. et al. *Building Evolutionary Architectures*. См. также Ford N. and Parsons R. Microservices as an Evolutionary Architecture. *ThoughtWorks*. March 11, 2016. - URL: https://www.

thoughtworks.com/insights/blog/microservices-evolutionary-architecture.

88. См. Helland P. *Data on the Outside versus Data on the Inside.*

89. Там же.

90. См. Event streaming as the source of truth. *ThoughtWorks.*

91. См. Kleppmann M. Turning the database inside-out with Apache Samza.

92. См. Calderwood B. Toward a Functional Programming Analogy for Microservices.

93. См. Hickey R. Deconstructing the Database.

94. См. Kreps J. The Log: What every software engineer should know about real-time data's unifying abstraction.

95. Nagle T. et al. Only 3% of Companies' Data Meets Basic Quality Standards. *Harvard Business Review*. September 11, 2017. - URL: https://hbr.org/2017/09/only-3-of-companies-data-meets-basic-quality-standards.

96. Infrastructure as code. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Infrastructure_as_code.

97. Sitakange J. Infrastructure as Code: A Reason to Smile. *ThoughtWorks*. March 14, 2016. - URL: https://www.thoughtworks.com/insights/blog/infrastructure-code-reason-smile.

98. См. Transforms. *Confluent.*

99. Kafka Streams Application Reset Tool. *Confluence*. - URL:https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Application+Reset+Tool. См. также Sax M. J. Data Reprocessing with the Streams API in Kafka: Resetting a Streams Application.

100. См. Master data management. *Wikipedia.*

101. Browne J. Brewer's CAP Theorem. *JulianBrowne.com*. January 11, 2009. - URL: http://www.julianbrowne.com/article/brewers-cap-theorem.

102. ACID. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/ACID.

103. Strong consistency. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Strong_consistency.

104. Eventual consistency. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Eventual_consistency.

105. Corbett J. et al. Spanner: Google's Globally-Distributed Database. *ACM Trans. Comput. Syst*. - URL: https://ai.google/research/pubs/pub39966.

106. X/Open XA. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/X/Open_XA.

107. Should I use XA transactions (two phase commit?). *Apache Active MQ*. - URL: http://activemq.apache.org/should-i-use-xa.html.

108. Curtiss M. Why you should pick strong consistency, whenever possible. *Google Cloud*. January 12, 2018. - URL: https://cloud.google.com/blog/products/gcp/why-you-should-pick-strong-consistency-whenever-possible.

109. Conflict-free replicated data type. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type.

110. Single Writer Principle. *Mechanical Sympathy*. September 22, 2011- URL: https://mechanical-sympathy.blogspot.com/2011/09/single-writer-principle.html.

111. Agha G. A. Actors: A Model of Concurrent Computation in Distributed Systems. *AI Technical Reports*. - URL: https://dspace.mit.edu/handle/1721.1/6952. См. также Actor model. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Actor_model.

112. Stonebraker M. et al. The End of an Architectural Era (It's Time for a Complete Rewrite). -

URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.3697&rep=rep1&type=pdf. См. также Stonebraker M. and Weisberg A. The VoltDB Main Memory DBMS. - URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.310.9242&rep=rep1&type=pdf.

113. Single responsibility principle. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Single_responsibility_principle.

114. См. ESB. *ThoughtWorks,* а также Recreating ESB antipatterns with Kafka. *ThoughtWorks.*

115. См. Helland P. *Data on the Outside versus Data on the Inside.*

116. Kung H.T. and Robinson J. T. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*. - URL: http://www.eecs.harvard.edu/~htk/publication/1981-tods-kung-robinson.pdf.

117. Two Generals' Problem. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Two_Generals'_Problem.

118. A Brief Tour of FLP Impossibility. *The Paper Trail*. August 13, 2008. - URL: https://www.the-paper-trail.org/post/2008-08-13-a-brief-tour-of-flp-impossibility/.

119. Idempotence. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Idempotence.

120. End-to-end principle. *Wikipedia*. - URL: hhttps://en.wikipedia.org/wiki/End-to-end_principle.

121. Colyer A. Distributed Snapshots: Determining Global States of Distributed Systems. *The Morning Paper*. April 22, 2015. - URL:https://blog.acolyer.org/2015/04/22/distributed-snapshots-determining-global-states-of-distributed-systems/.

122. Two-phase commit protocol. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Two-phase_commit_protocol.

123. KIP-98 perf testing. - URL: https://docs.google.com/spreadsheets/d/1dHY6M7qCiX-NFvsgvaE0YoVdNq26uA8608XIh_DUpI4/edit#gid=61107630.

124. Mehta A. and Gustafson J. Transactions in Apache Kafka. *Confluent*. November 17, 2017. - URL: https://www.confluent.io/blog/transactions-apache-kafka/.

125. См. X/Open XA. *Wikipedia.*

126. Stopford B. Chain Services with Exactly Once Guarantees. *Confluent*. July 26, 2017. - URL: https://www.confluent.io/blog/chain-services-exactly-guarantees/.

127. Garcaa-Molrna H. and Salem K. Sagas. *ACM*. - URL: http://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf.

128. Protocol Buffers. - URL: https://developers.google.com/protocol-buffers/.

129. JSON Schema. - URL: http://json-schema.org/.

130. Apache Avro. - URL: https://avro.apache.org/.

131. Schema Management. *Confluent*. - URL: https://docs.confluent.io/current/schema-registry/index.html.

132. Henson T. Schema On Read vs. Schema On Write Explained. November 14, 2016. - URL: https://www.thomashenson.com/schema-read-vs-schema-write-explained/.

133. Dead Letter Exchanges. - URL: https://www.rabbitmq.com/dlx.html. См. также Dead-letter queues. *IBM Knowledge Center*. - URL: https://www.ibm.com/support/knowledgecenter/SSFKSJ_7.1.0/com.ibm.mq.doc/ic10420_.htm.

134. Data degradation. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Data_degradation.

135. Xia N. Building Reliable Reprocessing and Dead Letter Queues with Apache Kafka. *Uber Engi-*

neering. February 16, 2018. - URL: https://eng.uber.com/reliable-reprocessing/.

136. Crypto-shredding. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Crypto-shredding.

137. Authorization and ACLs. *Confluent*. - URL: https://docs.confluent.io/current/kafka/authorization.html.

138. Sidecar pattern. *Microsoft Azure*. - URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar.

139. Waehner K. How to Build a UDF and/or UDAF in KSQL 5.0. *Confluent*. August 17, 2018. - URL: https://www.confluent.io/blog/build-udf-udaf-ksql-5-0.

140. Kafka-streams-examples. *GitHub*. - URL: https://github.com/confluentinc/kafka-streams-examples/tree/4.0.0-post/src/main/java/io/confluent/examples/streams/microservices.

141. Clients. *Confluence*. - URL: https://cwiki.apache.org/confluence/display/KAFKA/Clients.

142. Windowing. *Confluent*. - URL: https://docs.confluent.io/current/streams/concepts.html#windowing.

143. Streams Developer Guide. *Confluent*. - URL: https://docs.confluent.io/current/streams/developer-guide/index.html#state-stores.

144. Star schema. *Wikipedia*. - URL: https://en.wikipedia.org/wiki/Star_schema.

145. См. Kafka-streams-examples. *GitHub*.

146. Developer Guide for Kafka Streams. *Apache Kafka*. - URL: https://kafka.apache.org/documentation/streams/developer-guide/#streams_interactive_queries.

147. Join co-partitioning requirements. *Confluent*. - URL: https://docs.confluent.io/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl-joins-co-partitioning.

148. HDFS Snapshots. *Apache Hadoop*. - URL: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsSnapshots.html.

Научно-техническое издание

Отпечатано в ООО «Восточно-Сибирская типография»
Заказ № Усл. печ. л. Формат
664009, Иркутск, ул. Советская 109/3
Тел.: +7 (3952) 93-22-94, эл. почта: 932294@mail.ru