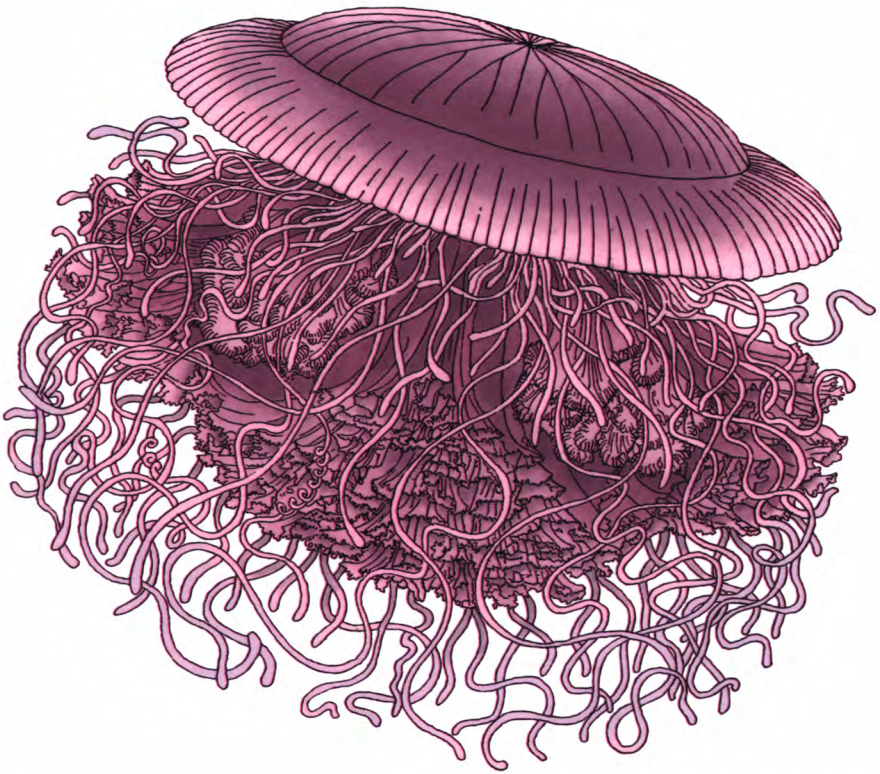
O'REILLY®

ОТ МОНОЛИТА К МИКРОСЕРВИСАМ

Эволюционные шаблоны для трансформации монолитной системы



Сэм Ньюмен

bhv®

Monolith to Microservices

Evolutionary Patterns to Transform Your Monolith

Sam Newman

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

Сэм Ньюмен

ОТ МОНОЛИТА К МИКРОСЕРВИСАМ

Санкт-Петербург
«БХВ-Петербург»
2021

УДК 004.43
ББК 32.973.26-02
Н93

Ньюмен С.

Н93 От монолита к микросервисам: Пер. с англ. — СПб.: БХВ-Петербург, 2021. — 272 с.: ил.

ISBN 978-5-9775-6723-7

Новая книга Сэма Ньюмена подробно описывает проверенный метод перевода существующей монолитной системы на микросервисы, поддерживающий работу организации в обычном режиме. Она дополняет его бестселлер «Создание микросервисов». Руководство содержит наглядные примеры, шаблоны миграции, массу практических советов по переводу монолитной системы на платформу для микрослужб, различные сценарии и стратегии успешной миграции, начиная с первичного планирования и заканчивая декомпозицией приложений и баз данных. Описанные шаблоны и методы можно использовать для миграции уже существующей архитектуры.

Для системных архитекторов, разработчиков и ИТ-специалистов

УДК 004.43
ББК 32.973.26-02

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Сависте*
Перевод с английского	*Андрея Логунова*
Компьютерная верстка	*Ольги Сергиенко*
Оформление обложки	*Карины Соловьевой*

Подписано в печать 13.01.21.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 21,93.
Тираж 1000 экз. Заказ № 006.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г Чехов, ул Полиграфистов, д 1

ISBN 978-1-492-04784-1 (англ.)
ISBN 978-5-9775-6723-7 (рус.)

Оглавление

Об авторе

Сэм Ньюмен — разработчик, архитектор, писатель и оратор, который работал с разными компаниями в разных областях по всему миру. Он работает независимо, концентрируя свое внимание в основном на облачных вычислениях, непрерывной доставке и микрослужбах. Его предыдущая книга — бестселлер "Создание микросервисов", также изданный в издательстве O'Reilly.

Когда он не "прыгает с одной подножки вагона на другую", его можно найти в сельской местности Восточного Кента, занимающимся различными видами спорта.

Предисловие

Еще несколько лет назад некоторые из нас лишь поговаривали о том, что, дескать, микрослужбы (микросервисы[1]) — интересная идея. И вот не успели мы оглянуться, как они стали архитектурой, принятой по умолчанию в сотнях компаний по всему миру (многие, вероятно, запущены как стартапы, призванные решать проблемы, вызванные микрослужбами), что заставило всех "перейти на бег", чтобы успеть "запрыгнуть на подножку последнего вагона", который, как они опасаются, вот-вот исчезнет за горизонтом.

Должен признаться, здесь есть часть моей вины. С тех пор как в 2015 году я написал свою собственную книгу "Создание микросервисов " (Building Microservices) на эту тему, я зарабатываю на жизнь, работая с людьми, помогая им понять данный тип архитектуры. Я всегда пытался сделать одно — прорваться сквозь хайп и помочь компаниям определиться, подходят ли им микрослужбы или нет. Для многих моих клиентов с существующими (не ориентированными на микрослужбы) системами трудность состояла в том, как внедрить архитектуры, основанные на микрослужбах. Как взять существующую систему и выполнить перепланировку ее архитектуры, не останавливая всю остальную работу? Вот где на помощь приходит эта книга. Что еще важнее, я постараюсь дать вам честную оценку трудностей, связанных с архитектурой на основе микрослужб, и помочь вам понять, стоит ли начинать это "путешествие".

Чему вы научитесь

Эта книга задумана как глубокое погружение в образ мыслей и порядок действий при разложении существующих систем на архитектуру, основанную на микрослужбах. Мы коснемся многих тем, связанных с архитектурой на основе микрослужб, но в центре внимания будет находиться декомпозиция. В качестве более общего руководства по архитектуре на основе микрослужб хорошим местом для старта была бы моя предыдущая книга "Создание микросервисов ". На самом деле я настоятельно рекомендую вам рассматривать ту книгу как дополнение к этой.

Глава 1 содержит общий обзор того, что такое микрослужбы, и далее разведаны идеи, которые привели нас к такого рода архитектуре. Этот материал будет хорошим подспорьем для новичков в микрослужбах, но я также настоятельно призываю даже более опытных разработчиков не пропускать эту главу. Чувствую, что в шквале технологий некоторые стержневые идеи микрослужб часто упускаются из виду: к этим концепциям книга будет возвращаться снова и снова.

[1] Далее "микрослужбы", см. *Комментарии переводчика* (после *Предисловия*).

Глубоко разбираться в микрослужбах совсем неплохо, но знать, подходят они вам или нет, — это нечто другое. В *главе 2* я расскажу, как оценить пригодность микрослужб для ваших условий, а также дам некоторые действительно важные рекомендации по управлению транзитом с монолита на архитектуру, основанную на микрослужбах. Здесь мы коснемся всего, от доменно-обусловленного дизайна до моделей организационных изменений — жизненно важных основ, которые окажут вам неоценимую помощь, даже если вы решите не использовать архитектуру на основе микрослужб.

В *главах 3* и *4* мы глубже погрузимся в технические аспекты, связанные с декомпозицией монолита, изучая реальные примеры и выделяя шаблоны миграции. *Глава 3* сосредоточена на аспектах декомпозиции приложений, а *глава 4* представляет собой глубокое погружение в вопросы, связанные с данными. Если вы действительно хотите перейти с монолитной системы на архитектуру, основанную на микрослужбах, то вам придется разобрать некоторые базы данных на части!

Наконец, в *главе 5* рассматриваются всякого рода трудности, с которыми вы столкнетесь по мере роста архитектуры на основе микрослужб. Эти системы способны принести огромную выгоду, но они также сопровождаются большой сложностью и проблемами, с которыми вам не приходилось сталкиваться раньше. Эта глава является моей попыткой помочь вам засечь такие проблемы, когда они только начинают возникать, и предложить способы борьбы с болезнями роста, связанными с микрослужбами.

Условные обозначения, принятые в книге

В книге используются следующие типографские условные обозначения:

- *Курсив* — указывает новые термины, URL-адреса, адреса электронной почты, имена файлов и расширения файлов.
- `Моноширинный шрифт` — применяется для листингов программ, а также внутри абзацев для ссылки на элементы программ, такие, как переменные или имена, инструкции языка и ключевые слова.
- **`Полужирный моноширинный шрифт`** — выделяет команды либо другой текст, который должен быть напечатан пользователем буквально.
- *`Моноширинный шрифт курсивом`* — показывает текст, который должен быть заменен значениями пользователя либо значениями, определяемыми по контексту.

Данный элемент обозначает подсказку или совет.

Данный элемент обозначает общее замечание.

Данный элемент обозначает предупреждение или предостережение.

На веб-странице книги перечислены ошибки, приведены примеры и дополнительная информация. Вы можете обратиться к этой странице по адресу:

https://oreil.ly/monolith-to-microservices.

Благодарности

Без помощи и понимания моей замечательной жены Линди Стивенс эта книга была бы невозможной. Эта книга для нее. Линди, прости, что я так ворчал, когда приближались и проходили разные сроки завершения. Я также хотел бы поблагодарить прекрасный клан Гиллман Стейнс за всю их поддержку — мне повезло, что у меня такие замечательные родные.

Эта книга во многом выиграла благодаря людям, которые любезно пожертвовали свое время и энергию на то, чтобы прочитать различные черновики и дать ценные идеи. Я особенно хочу поблагодарить Криса О'Делла, Дэниела Брайанта, Пита Ходжсона, Мартина Фаулера, Стефана Шрасса и Дерека Хаммера за их усилия в этом деле. Были также люди, которые непосредственно внесли свой вклад во многих отношениях, поэтому я также хотел бы здесь поблагодарить Грэма Тэкли, Эрика Доэрненберга, Марчина Засепу, Майкла Фетера, Рэнди Шоуп, Кифа Моррис, Питера Гилларда-Мосса, Мэтта Хита, Стива Фримена, Рене Ленгвината, Сару Уэллс, Риса Эванса и Берка Сохана. Если вы найдете ошибки в этой книге, то эти ошибки будут моими, а не их.

Коллектив издательства O'Reilly также оказал невероятную поддержку, и я хотел бы высоко оценить труд моих редакторов Элеоноры Брю и Алисии Янг, в дополнение к Кристоферу Гузиковски, Мэри Трезелер и Рейчел Румелиотис. Также хочу сказать большое спасибо Хелен Кодлинг и ее коллегам по всему миру за то, что они продолжают таскать мои книги на различные конференции, Сьюзен Конант за то, что она помогает мне оставаться в здравом уме, ориентируясь в меняющемся мире издательского дела, и Майку Лукидесу за то, что он изначально привлек меня к работе с O'Reilly. Я знаю, что есть еще много людей "за кулисами", которые оказывали помощь, так что спасибо вам всем.

Помимо тех, кто непосредственно внес свой вклад в эту книгу, также хочу назвать и тех, кто осознанно или нет, помог этой книге появиться. Поэтому хотел бы поблагодарить (в произвольном порядке) Мартина Келппманна, Бена Стопфорда, Чарити Мейджорс, Алистера Кокберна, Грегора Хупе, Бобби Вулфа, Эрика Эванса, Ларри Константина, Лесли Лэмпорта, Эдварда Йордона, Дэвида Парнаса, Майка Блэнда, Дэвида Вудса, Джона Оллспоу, Альберто Брандолини, Фредерика Брукса, Синди Сридхаран, Дейва Фарли, Джез Хамбл, Джина Кима, Джеймса Льюиса, Николь Форсгрен, Гектора Гарсиа-Молину, Sheep & Cheese, Кеннет Салема, Адриан Кольера, Пэе Хелланд, Крестен Торупа, Хенрика Книберга, Андерса Иварссона, Мануэль Пайс, Стива Смита, Бернда Ракера, Мэтью Скелтона, Алексиса Ричардсона, Джеймса Говернера и Кейн Стивенс.

Как это всегда бывает в подобных ситуациях, мне кажется весьма вероятным, что я упустил кого-то, кто внес в эту книгу существенный вклад. Этим людям я могу ска-

зать только одно: мне очень жаль, что я забыл поблагодарить вас лично, и я надеюсь, что вы сможете простить меня.

Наконец, время от времени некоторые спрашивают меня об инструментах, которые мне потребовались для написания этой книги. Я писал в AsciiDoc, используя код Visual Studio вместе с плагином AsciiDoc, построенным Джоао Пинто (João Pinto). Книга находилась под контролем системы управления версиями Git, с системой Atlas O'Reilly. Я писал в основном на своем ноутбуке с внешней механической клавиатурой Razer, но ближе к концу также активно использовал iPad Pro с Git-клиентом Working Copy, для того чтобы закончить последние несколько разделов. Это позволило мне писать во время путешествия, обеспечив возможность в одном памятном случае писать о рефакторизации базы данных на пароме до Оркнейских островов. Вызванная этим морская болезнь стоила того.

Комментарии переводчика

Служба или сервис?

В настоящем переводе за основу принят широко известный методологический принцип "Бритвы Оккама", согласно которому следует избегать многообразия в терминологии без крайней на то необходимости (он формулируется и по-другому — не следует порождать "пустых" сущностей).

В ситуации, когда отсутствуют стандарты перевода научно-технических терминов, основными критериями служат авторитетные источники информации, каковыми являются Википедия[1], как унифицирующий (и нужно признать, иногда не бесспорный) источник знаний, и гиганты ИТ-индустрии.

Такие глобальные компании ИТ-индустрии, как Microsoft и IBM (их российские филиалы) в своей документации придерживаются перевода и интерпретации английского термина "microservice", именно как микрослужбы[2,3].

Микрослужбами называют тысячи независимых веб-стандартов, языков программирования, платформ баз данных и компонентов веб-серверов, используемых в современном жизненном цикле разработки ПО в качестве средств разработчиков. Организации, применявшие традиционный подход, предпочитали архитектуру, ориентированную на услуги, которая представляла собой сочетание оборудования и ПО, предоставляемых одной ИТ-компанией. Микрослужбы обеспечивают поддержку тысяч различных компонентов, предоставляемых независимыми компаниями по разработке или сообществами по созданию решений с открытым исходным кодом, в облачных приложениях и на веб-серверах. ИТ-отделам требовался новый подход к управлению микрослужбами в производственной инфраструктуре, состоящей из изолированных многоарендных сред в гипермасштабных центрах обработки данных (ЦОД) на базе публичных облаков. В связи с этим многие ИТ-отделы внедряли решения по виртуализации со стандартами программно-определяемого ЦОД на базе технологии Service Mesh. Микрослужбы представляют собой структурные блоки или основные компоненты, платформы и структуры, на базе которых создается и выполняется программный код на веб-серверах в облачном ЦОД.

---

[1] См. **https://ru.wikipedia.org/wiki/Веб-служба**.

[2] См. **https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices**.

[3] См. **https://www.ibm.com/developerworks/ru/library/cl-bluemix-microservices-in-action-part-1-trs/**.

Следует добавить еще одну важную для понимания деталь. В информатике со времен ее формирования в 1960-х годах сложились определенные традиции перевода технических терминов, и с тех времен ничего не поменялось; изменились лишь люди, которые в последнее время все больше употребляют англицизмы, что очень часто затуманивает смысл и суть дела. Каждый термин принадлежит иерархии терминов, составляющей так называемое "древо знаний". Когда мы говорим "шаблон архитектурного дизайна" (pattern), мы четко определяем место этого термина в указанной иерархии. С другой стороны, перевод "паттерн" эту связь обрывает. То же касается, скажем, термина "каркас" (framework), который в зависимости от контекста, может быть вычислительным или математическим каркасом программирования и разработки. И, опять-таки, перевод "фреймворк" помимо своей непроизносимости на русском эту связь обрывает. Все указанное относится и к термину "microservice".

По всем этим причинам в настоящей книге термин "microservice" переведен как "микрослужба".

ГЛАВА 1

Основные сведения о микрослужбах

Скажем так, все обострилось в один миг и быстро вышло из-под контроля!
– *Рон Бургунди, Ведущий*

Прежде чем мы углубимся в приемы работы с микрослужбами, важно, чтобы у нас было общее, совместное понимание того, что такое архитектура на основе микрослужб. Я хотел бы обратиться к некоторым распространенным заблуждениям, которые встречаю регулярно, а также к нюансам, которые часто упускаются. Вам понадобится этот прочный фундамент знаний, чтобы извлечь максимальную пользу из остальной части книги. В данной главе приведено объяснение архитектуры на основе микрослужб, кратко рассказано о том, как микрослужбы развивались (что, естественно, означает рассмотрение монолитов), а также дан анализ некоторых преимуществ и трудностей работы с микрослужбами.

Что такое микрослужбы?

Микрослужбы — это независимо развертываемые службы, моделируемые вокруг бизнес-домена. Они общаются друг с другом через сети и предлагают на выбор целый ряд вариантов архитектуры для решения задач, с которыми вы можете столкнуться. Отсюда следует, что архитектура на основе микрослужб базируется на многочисленных сотрудничающих микрослужбах.

Рассматриваемый архитектурный тип ориентирован на службы (service-oriented architecture, SOA); не углубляясь в вопрос о том, как должны очерчиваться контуры служб, отметим, что ключом является их независимая развертываемость. Преимущество микрослужб также в том, что они нейтральны к технологиям.

С технологической точки зрения микрослужбы выставляют наружу возможности бизнеса, которые они инкапсулируют через одну или несколько конечных точек в сети. Микрослужбы взаимодействуют друг с другом через эти сети, что делает их разновидностью распределенной системы. Они также инкапсулируют хранение и извлечение данных, предоставляя данные через четко определенные интерфейсы. И поэтому базы данных скрыты в пределах контура службы.

Здесь есть много чего, в чем следует разобраться, поэтому давайте слегка углубимся в некоторые из этих идей.

Независимая развертываемость

Независимая развертываемость — это идея о том, чтобы вносить изменения в микрослужбу и развертывать ее в производственной среде без необходимости использовать какие-либо другие службы. Что еще важнее, дело не просто в том, что мы *можем* это осуществить, а в том, как вы управляете развертываниями в своей системе *на самом деле*. Это дисциплина, которую вы практикуете в большинстве ваших релизов. Это простая идея, которая, тем не менее, сложна в исполнении.

Если из этой книги вы извлечете только одну полезную вещь, то она должна быть такой: обеспечить внедрение концепции независимой развертываемости микрослужб. Приобретите привычку вносить изменения в свое производство в отдельной микрослужбе без развертывания чего-либо еще. Из этого последует много хорошего.

Для обеспечения гарантии независимой развертываемости наши службы должны быть слабо сопряжены — другими словами, мы должны иметь возможность изменять одну службу без необходимости менять что-либо еще. Это означает, что нам нужны ясные, четко определенные и стабильные контракты между службами. Некоторые варианты имплементации это затрудняют, например, использование совместных баз данных является особенно проблемным. Стремление к слабо сопряженным службам со стабильными интерфейсами направляет наше мышление прежде всего на отыскание контуров служб.

Моделируются вокруг бизнес-домена

Внесение изменения через контур процесса обходится дорого. Если для внедрения функции вам нужно изменить две службы и выполнить оркестровку указанных двух изменений, то это займет больше времени, чем внесение одинакового изменения внутри одной службы (или, если на то пошло, монолита). Из этого следует, что мы хотим обеспечить, чтобы трансграничные изменения между службами вносились нами как можно реже.

Следуя тому же подходу, который я употребил в книге "Создание микросервисов", в этой книге используется поддельный домен и фиктивная компания с целью иллюстрации некоторых концепций, когда невозможно поделиться реальными историями. Речь идет о компании Music Corp, крупной многонациональной организации, которая так или иначе остается в бизнесе, несмотря на то что она почти полностью сосредоточена на продаже компакт-дисков.

Упираясь руками и ногами, мы решили перенести Music Corp в XXI век, и в рамках этой компании мы оцениваем существующую системную архитектуру. На рис. 1.1 мы видим простую трехслойную архитектуру. У нас есть веб-интерфейс пользователя, слой бизнес-логики в форме монолитного бэкэнда (серверной части) и хранилища данных в традиционной базе данных. Эти слои, как обычно, принадлежат разным операционным группам.

Мы хотим внести в нашу функциональность простое изменение: нам нужно, чтобы наши заказчики могли указывать свой любимый жанр музыки. При этом потребу-

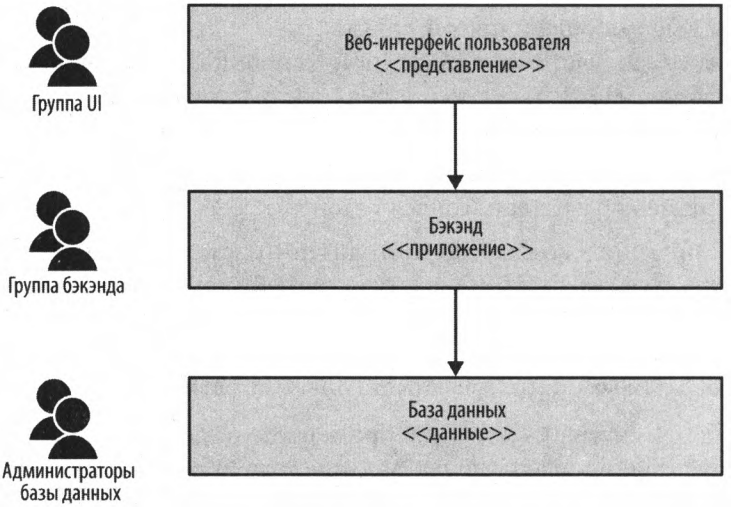


Рис. 1.1. Системы компании Music Corp как традиционная трехслойная архитектура

ется изменить пользовательский интерфейс (UI), показывая в нем жанры на выбор, чтобы бэкэнд обеспечивал появление жанра в UI и изменение значения и чтобы база данных принимала это изменение. Эти изменения должны управляться каждой группой, как показано на рис. 1.2, и они должны быть развернуты в правильном порядке.

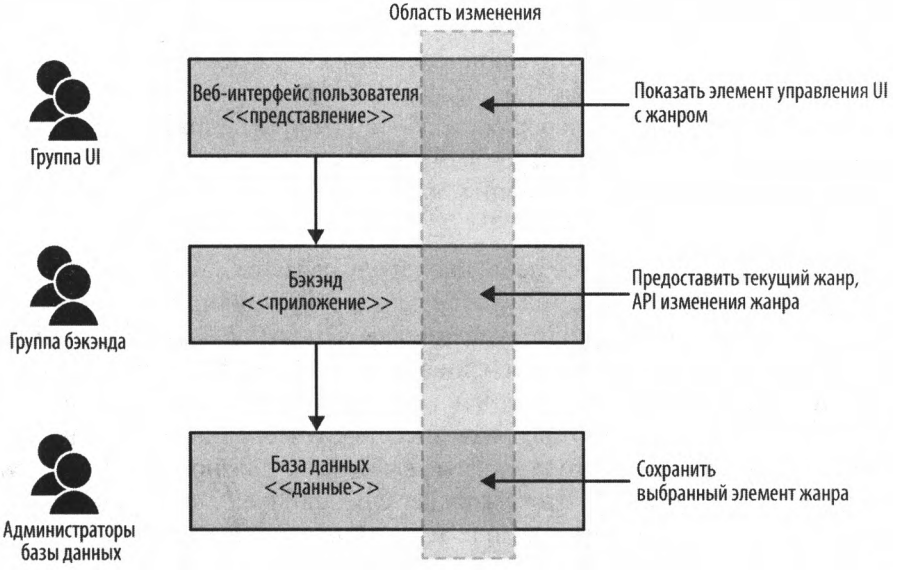


Рис. 1.2. Вносить изменения во всех трех слоях оказывается труднее

Теперь эта архитектура является не такой уж плохой. Вся архитектура в итоге оптимизируется вокруг некоторого множества целей. Трехъярусная архитектура настолько распространена отчасти потому, что она универсальна — все о ней слы-

шали. Так что выбор общепринятой архитектуры, которую вы, возможно, видели где-то в другом месте, часто является одной из причин, по которой мы продолжаем видеть этот шаблон. Но я думаю, что самая веская причина, почему мы встречаем эту архитектуру снова и снова, связана с тем, что она основана на способе организации наших групп разработчиков.

Ставший уже знаменитым закон Конвея гласит:

> "Любая организация, которая строит дизайн системы... неизбежно произведет дизайн, структура которого является копией коммуникационной структуры организации".
>
> – *Мелвин Конвей (Melvin Conway)*,
> Каким образом комитеты изобретают (How Do Committees Invent)?

Трехъярусная архитектура — хороший пример этого закона в действии. В прошлом ИТ-организации группировали людей в соответствии с их стержневыми компетенциями: администраторы баз данных — в группе с другими администраторами баз данных; разработчики на Java — в группе с другими разработчиками на Java; и разработчики фронтэнда (фронтальной части) (которые в настоящее время знают такие экзотические вещи, как JavaScript и разработка собственных мобильных приложений) были еще в одной группе. Мы группируем людей на основе их стержневых компетенций, поэтому мы создаем ИТ-активы, которые выравниваются по этим группам.

Указанный факт объясняет, почему эта архитектура так распространена. Она неплохая; она просто оптимизирована вокруг одного множества сил — так же как мы традиционно группируем людей — вокруг дружеских отношений. Но силы изменились. Наши стремления относительно программно-информационного обеспечения[1] изменились. Мы теперь объединяем людей в поликвалифицированные группы с целью сократить эстафетные передачи и обособленные подразделения. Мы хотим осуществлять доставку софта гораздо быстрее, чем когда-либо прежде. Это заставляет нас принимать различные решения, касающиеся организации наших групп и разделения наших систем на части.

Изменения в функциональности в первую очередь связаны с изменениями в бизнес-функциональности. Но на рис. 1.1 наша бизнес-функциональность фактически распределена по всем трем слоям, увеличивая вероятность того, что изменение функциональности приведет к пересечению слоев. В такой архитектуре мы имеем высокую связность родственных технологий, но низкую связность бизнес-функциональности. Если мы хотим упростить внесение изменений, то нам нужно изменить способ группирования кода — мы выбираем связность бизнес-функциональности, а не технологии. В итоге каждая служба может и не содержать смесь этих трех слоев, но это уже вопрос имплементации локальной службы.

---

[1] Понятия "софт" и "программно-информационное обеспечение" в переводе используются взаимозаменяемо. Последний термин используется именно в такой формулировке, исходя из зарубежной трактовки термина software, как "программ и операционной информации, необходимых компьютеру" (ср. **https://en.wikipedia.org/wiki/Software**), т. е. софт — это логика и данные, которыми оперирует компьютер — *Пер.*

Давайте сравним эту архитектуру с потенциальной альтернативой, показанной на рис. 1.3. У нас есть специализированная служба "Клиенты", которая предоставляет UI, позволяющий клиентам обновлять свою информацию, при этом состояние клиента также хранится в той же службе. Выбор любимого жанра ассоциирован с конкретным клиентом, поэтому данное изменение гораздо более локализовано. На рис. 1.3 мы также показываем список имеющихся жанров, извлекаемых из службы "Каталог", вероятно, что-то, что уже там имелось. Мы также видим, что новая служба "Рекомендации" обращается к нашей информации о любимом жанре, чему-то, что легко последует в дальнейшем релизе.

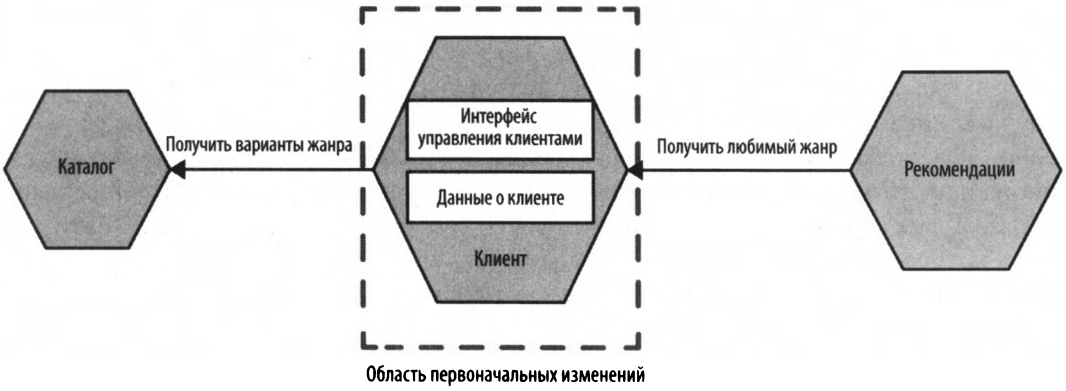


Рис. 1.3. Специализированная служба "Клиент" значительно облегчает регистрацию любимого музыкального жанра клиента

В такой ситуации наша служба "Клиент" инкапсулирует тонкую "дольку" каждого из трех уровней: она имеет немножечко UI, немножечко прикладной логики и немножечко хранения данных, но все эти слои инкапсулированы в одной службе.

В итоге бизнес-домен становится первостепенной силой, движущей нашу системную архитектуру, которая, надо надеяться, облегчит внесение изменений и упростит организацию наших групп вокруг бизнес-домена. Это представляется настолько важным, что прежде чем мы закончим эту главу, мы вернемся к концепции моделирования программно-информационного обеспечения вокруг домена, благодаря чему я смогу поделиться некоторыми идеями о доменно-обусловленном дизайне — идеями, которые формируют наше представление об архитектуре на основе микрослужб.

Владеют своими собственными данными

Одна из идей, с которыми, на моем опыте, люди испытывают особые трудности, состоит в том, что микрослужбы не должны использовать базы данных совместно. Если одна служба хочет обратиться к данным, хранящимся в другой службе, то она должна пойти и запросить у этой службы необходимые ей данные. Такой подход дает той службе возможность решать, что является совместным, а что скрытым. Он также позволяет службе отображать детали внутренней имплементации, которые

могут измениться по различным произвольным причинам, в более четкий публичный контракт, обеспечивающий стабильные интерфейсы между службами. Наличие стабильных интерфейсов между службами имеет большое значение, если мы хотим независимой развертываемости. Если интерфейс, который служба выставляет наружу, продолжает изменяться, то это изменение создаст "волновой эффект", заставляющий изменяться другие службы.

Не предоставляйте совместный доступ к базам данных, если вам это не нужно в действительности. Постарайтесь сделать все возможное, чтобы избежать его. На мой взгляд, это одна из худших вещей, которую вы сделаете, если попытаетесь достичь независимой развертываемости.

Как уже обсуждалось в предыдущем разделе, мы хотим рассматривать наши службы как сквозные "куски" бизнес-функциональности, которые в соответствующих случаях инкапсулируют UI, прикладную логику и хранение данных. Это обусловлено желанием уменьшить усилия, необходимые для изменения бизнес-функциональности и всего, что с ней связано. Инкапсуляция данных и поведения таким образом дает нам высокую связность бизнес-функциональности. Скрывая базу данных, которая поддерживается нашей службой, мы также обеспечиваем снижение сопряженности. Мы еще вернемся к сопряженности и связности чуть позже.

Это бывает трудно понять, в особенности, когда вы имеете существующую монолитную систему с гигантской базой данных, с которой вам приходится работать. К счастью, *глава 4* полностью посвящена отказу от монолитных баз данных.

Какие преимущества приносят микрослужбы?

Преимущества микрослужб многочисленны и разнообразны. Независимая природа развертываний открывает новые модели повышения масштаба и робастности систем, а также позволяет комбинировать и сочетать технологии. Поскольку над службами можно работать параллельно, вы сможете привлекать к задаче больше разработчиков, без того чтобы они вставали друг у друга на пути. Кроме того, этим разработчикам станет проще понять свою часть системы, т. к. они сосредоточивают свое внимание только на одной ее части. Изоляция процессов также позволяет нам варьировать выбор технологий, возможно, смешивая разные языки программирования, стили программирования, платформы развертывания или базы данных в поисках их правильной смеси.

Помимо всего прочего, архитектуры на основе микрослужб, пожалуй, дают вам гибкость. Они открывают гораздо больше возможностей относительно того, как решать задачи в будущем.

Однако важно отметить, что ни одно из этих преимуществ не дается бесплатно. Существует много подходов к декомпозиции системы, и изначально преследуемая вами цель будет вести эту декомпозицию в разных направлениях. Следовательно, становится важным понимание вопроса о том, что вы пытаетесь получить от архитектуры на основе микрослужб.

Какие проблемы они создают?

Ориентированная на службы архитектура вошла в моду отчасти потому, что компьютеры стали дешевле, и поэтому их стало больше. Вместо того чтобы развертывать системы на одном гигантском мэйнфрейме, разумнее использовать несколько более дешевых машин. Архитектура, ориентированная на службы, была попыткой выработать наилучший способ строительства приложений, охватывающих многочисленные машины. Одна из главных трудностей во всем этом заключается в тех способах, которыми указанные компьютеры "разговаривают" друг с другом: сетях.

Коммуникация между компьютерами по сетям не является мгновенной (это очевидным образом следует из физики). Отсюда ясно, что мы должны беспокоиться о латентностях — и в частности, о латентностях, намного превышающих те, которые мы видим у локальных, внутрипроцессных операций. Ситуация ухудшается, когда мы учтем, что эти латентности варьируются, делая поведение системы непредсказуемым. И мы также должны учитывать тот факт, что сети иногда выходят из строя — пакеты теряются; сетевые кабели отсоединяются.

Эти препятствия намного затрудняют действия, которые относительно просты с однопроцессным монолитом. И затрудняют настолько, что по мере усложнения системы вам, скорее всего, придется избавиться от транзакций и безопасности, которую они приносят, в обмен на другие виды методов (которые, к сожалению, имеют очень разные компромиссы).

Осознание того факта, что любой сетевой вызов может быть и будет безуспешным, становится головной болью, как и тот факт, что службы, с которыми вы обмениваетесь, по какой-либо причине могут отключиться или же начать вести себя странно. Вдобавок ко всему этому, вам также придется приступить к попыткам выработать способ получения согласованного представления данных на многочисленных машинах.

И потом, конечно же, у нас есть огромный массив новых, дружественных для микрослужб технологий, которые следует принять в расчет. Если их использовать неумело, то они помогут вам совершать ошибки гораздо быстрее и более интересными, дорогостоящими способами. Честно говоря, микрослужбы выглядят ужасной идеей, если не считать всего хорошего.

Стоит отметить, что практически все системы, которые мы относим к категории "монолитов", также являются распределенными системами. Однопроцессное приложение, скорее всего, считывает данные из базы данных, работающей на другой машине, и далее передает данные в веб-браузер. Это, по крайней мере, смесь из трех компьютеров с коммуникацией между ними по сети. Разница состоит в степени, в которой монолитные системы "распределены" по сравнению с архитектурами на основе микрослужб. Чем больше будет компьютеров, которые общаются по большему количеству сетей, тем с большей вероятностью вы столкнетесь с неприятными проблемами, ассоциированными с распределенными системами. Эти кратко описанные мной проблемы, возможно, не появятся изначально, но со временем, по мере роста вашей системы, вы, вероятно, столкнетесь с большинством, если не со всеми из них.

Как сказал мой старый коллега, друг и эксперт по микрослужбам Джеймс Льюис (James Lewis) "микрослужбы покупают вам варианты". Когда Джеймс высказал эту мысль, он знал, что говорит — они *покупают* вам *варианты*. У них есть стоимость, и вы должны решить, сопоставима ли стоимость с ценой вариантов, за которые вы хотите ухватиться. Мы разведаем эту тему подробнее в *главе 2*.

Пользовательские интерфейсы

Слишком часто я вижу, что люди сосредоточивают свою работу на внедрении микрослужб исключительно на стороне сервера, оставляя пользовательский интерфейс (UI) как единый, монолитный слой. Если нам нужна архитектура, облегчающая более быстрое развертывание новых функций, то оставлять UI в виде большого монолитного объекта будет большой ошибкой. Мы также можем и должны подумать о том, чтобы разбить наши пользовательские интерфейсы на части, чем мы и займемся в *главе 3*.

Технология

Иногда возникает заманчивое искушение — ухватиться за целую связку новых технологий, движущихся в фарватере вашей совершенно новой архитектуры на основе микрослужб, но я настоятельно призываю вас не поддаваться этому искушению. Внедрение любой новой технологии будет иметь свою цену — оно создаст некий беспорядок. Надо надеяться, что оно будет стоить того (если вы выбрали правильную технологию, конечно!), но при первичном принятии архитектуры на основе микрослужб у вас будет происходить немало других вещей.

Выработка способа правильного эволюционного развития и управления архитектурой на основе микрослужб предусматривает обуздание целого ряда трудностей, связанных с распределенными системами, трудностей, с которыми вы, возможно, не сталкивались раньше. Думаю, что гораздо полезнее сначала разобраться в этих вопросах, по мере того как вы с ними сталкиваетесь, используя стек технологий, с которым вы знакомы, а затем подумать о том, поможет ли изменение существующей технологии решить эти проблемы по мере их обнаружения.

Как мы уже говорили, микрослужбы по своей сути нейтральны к технологиям. Пока ваши службы способны общаться друг с другом через сеть, все остальное можно "хватать и уносить". И в этом огромное преимущество, т. к. вы, если захотите, сможете смешивать и сочетать стеки технологий.

От вас не требуется использовать Kubernetes, Docker, контейнеры или публичное облако. От вас не требуется кодировать на Go или Rust или чем-то еще. На самом деле подбор языка программирования совершенно не важен в том, что касается архитектур на основе микрослужб, помимо того что некоторые языки имеют более богатую экосистему поддерживающих библиотек и вычислительных каркасов. Если вы знаете PHP лучше всего, то начните строить службы с PHP![2] По отноше-

[2] Для получения дополнительной информации по этой теме я рекомендую книгу Лорны Джейн Митчелл "Веб-службы PHP" (PHP Web Services, Lorna Jane Mitchell, O'Reilly).

нию к некоторым стекам технологий существует слишком много технического снобизма, который порой непростительно граничит с презрением к людям, работающим с теми или иными инструментами[3]. Не будьте частью этой проблемы! Выберите подход, который работает на вас, и меняйте вещи ради решения проблем, тогда и по мере того как вы их видите.

Размер

Вопрос, "насколько большой должна быть микрослужба?", наверное, самый распространенный из тех, которые я получаю. Учитывая, что приставка "микро" находится прямо в названии, ответ не станет удивительным. Однако если копнуть относительно того, что заставляет микрослужбы работать как архитектурный тип, то понятие размера на самом деле становится одним из наименее интересных.

Как измерять размер? Строками кода? Для меня это не имеет особого смысла. То, для чего на Java потребуется 25 строк кода, возможно, будет написано в 10 строках на Clojure. Это не значит, что Clojure лучше или хуже Java, скорее, некоторые языки выразительнее других.

По моему мнению, ближе всего к определению "размера", имеющего какой-то смысл с точки зрения микрослужб, были слова, однажды высказанные экспертом по микрослужбам Крисом Ричардсоном (Chris Richardson), заявившим, что цель микрослужб — иметь "минимально возможный интерфейс". Это определение согласуется с концепцией сокрытия информации (о которой мы поговорим чуть позже), но представляет собой попытку найти смысл постфактум. Когда мы впервые говорили об этих вещах, в центре нашего внимания, по крайней мере, изначально было то обстоятельство, что эти вещи действительно легко заменяются.

В конечном счете, понятие "размер" является весьма контекстуальным. Поговорите с человеком, который работал над системой в течение 15 лет, и по его ощущениям их система из 100K строк кода действительно будет простой для понимания. Спросите мнение кого-то, кто для этого проекта является абсолютным новичком, и по его ощущениям проект будет, скажем так, просто огромным. Точно так же спросите компанию, которая только что приступила к транзиту на микрослужбы, возможно, с десятью микрослужбами или меньше того, и вы получите другой ответ, чем от компании аналогичного размера, в которой микрослужбы были нормой в течение многих лет, а теперь они имеют их сотни.

Я призываю людей не беспокоиться о размере. Когда вы только начинаете, гораздо важнее сосредоточиться на двух ключевых вещах. Прежде всего, со сколькими микрослужбами вы справитесь? По мере того как у вас будет служб все больше и больше, сложность вашей системы будет увеличиваться, и вам придется усваивать новые навыки (и, возможно, внедрять новые технологии), для того чтобы с ней справляться. Именно по этой причине я решительно выступаю за поступательную

[3] Прочитав блог-пост Ауринн Шоу (Aurynn Shaw) "Культура презрения" (Contempt Culture, **http://bit.ly/2oeICgL**), я осознал, что в прошлом был виноват, проявляя некоторую степень презрения к разным технологиям и, соответственно, к сообществам вокруг них.

миграцию на архитектуру, основанную на микрослужбах. Во-вторых, как определить контуры микрослужб так, чтобы получить максимальную от них отдачу, не превращая все в ужасно сопряженный беспорядок? Эти темы мы рассмотрим в следующих разделах данной главы.

История термина "микрослужбы"

В 2011 году, когда я еще работал в консалтинговой компании ThoughtWorks, мой друг, а затем коллега Джеймс Льюис (James Lewis) заинтересовался тем, что он называл "микроприложениями". Он заметил, что этот шаблон применяется несколькими компаниями, которые использовали ориентированную на службы архитектуру, — они занимались оптимизацией этой архитектуры с целью сделать службы легко заменяемыми. Компании, о которых идет речь, были заинтересованы в быстром развертывании конкретной функциональности, но с тем расчетом, что ее можно будет переписать в других стеках технологий, когда это понадобится для масштабирования.

В то время особняком стоял вопрос, насколько эти службы были малы по объему. Некоторые из служб можно было написать (или переписать) в течение нескольких дней. Джеймс продолжал повторять, что "службы должны быть не больше, чем моя голова", причем идея заключалась в том, что объем функциональности должен быть таким, чтобы ее было легко понимать и, следовательно, легко изменять.

Позже, в 2012 году, Джеймс поделился этими идеями на архитектурном саммите, где присутствовали некоторые из разработчиков. На том семинаре мы обсуждали тот факт, что на самом деле эти приложения не были самодостаточными, поэтому название "микроприложения" было не совсем правильным. Вместо него более подходящим выглядел термин "микрослужбы"[4].

И владение

С помощью микрослужб, смоделированных вокруг бизнес-домена, мы видим выравнивание между нашими ИТ-артефактами (нашими независимо развертываемыми микрослужбами) и нашим бизнес-доменом. Эта идея хорошо резонирует, когда мы учитываем сдвиг в сторону разрушения разделяющих границ между "Бизнесом" и "ИТ" в технологических компаниях. В традиционных ИТ-организациях процесс разработки софта часто осуществляется частью организации, совершенно отличной от той, которая фактически определяет требования и имеет связь с клиентом, как показано на рис. 1.4. Дисфункции этих видов организаций многочисленны и разнообразны и, вероятно, не нуждаются в дальнейшем рассмотрении.

Вместо этого мы видим, как истинные технологические организации тотально объединяют эти предыдущие разрозненные организационные структуры, как показано на рис. 1.5. Владельцы продуктов теперь работают непосредственно в рамках групп доставки, причем эти группы выравниваются не вокруг произвольных технических группировок, а вокруг ориентированных на клиента продуктовых линий. Теперь нормой являются не централизованные ИТ-функции, а, наоборот, существование

[4] Не могу вспомнить, когда мы впервые написали этот термин, но я живо помню свое настойчивое требование, несмотря на всю логику относительно грамматики, что указанный термин не должен быть с дефисом. Оглядываясь назад, эту позицию трудно оправдать, но я придерживался ее, несмотря ни на что. Я настаиваю на своем неразумном, но, в конечном счете, победоносном варианте.

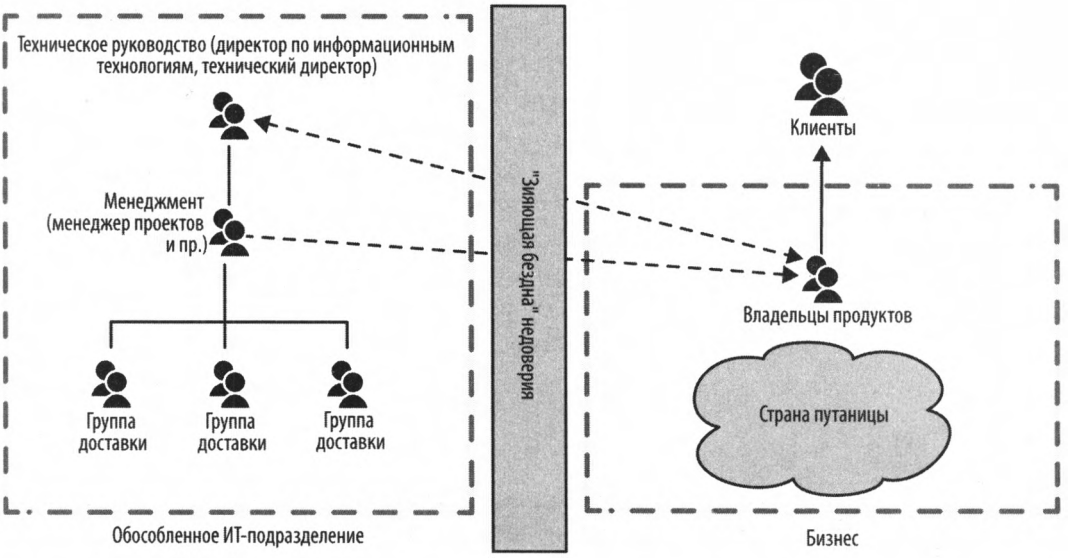


Рис. 1.4. Организационный вид традиционного деления на ИТ и Бизнес

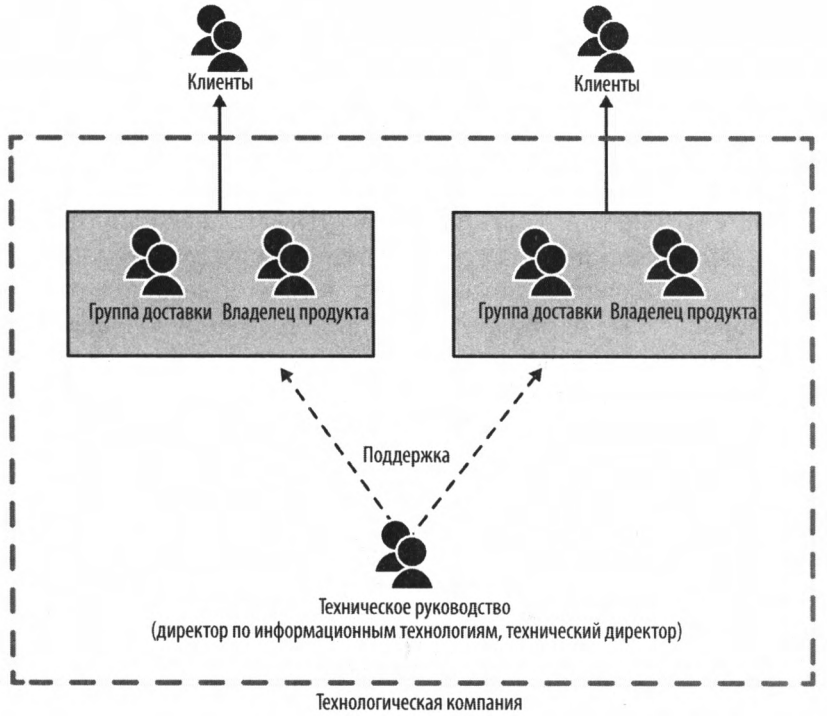


Рис. 1.5. Пример того, как истинные технологические компании интегрируют программно-информационные продукты

любой центральной ИТ-функции заключается в поддержке групп доставки, ориентированных на клиента.

Хотя не все организации совершили этот сдвиг, архитектуры на основе микрослужб намного упрощают такое изменение. Если вы хотите, чтобы группы доставки были выровнены по продуктовым линиям, а службы были выровнены по бизнес-домену, то становится проще четко предоставлять владение этим ориентированным на продукт группам доставки. Сокращение числа служб, совместно используемых многочисленными группами, является ключом к минимизации конкуренции за доставку. Архитектуры на основе микрослужб, ориентированные на бизнес-домен, значительно упрощают этот сдвиг в организационных структурах.

Монолит

Мы говорили о микрослужбах, но в этой книге речь идет о переходе от монолитов к микрослужбам, поэтому нам также нужно определиться с тем, что подразумевается под термином "монолит".

Когда в этой книге я говорю о монолитах, то в первую очередь имею в виду единицу развертывания. Когда все функциональности в системе должны разворачиваться вместе, мы считаем эту систему монолитом. Существует, по крайней мере, три типа монолитных систем, которые отвечают всем требованиям: однопроцессная система, распределенный монолит и сторонние черно-ящичные системы.

Однопроцессный монолит

Наиболее распространенный пример, который приходит на ум при обсуждении монолитов, — система, в которой весь код развертывается как один процесс, как показано на рис. 1.6. Можно иметь несколько экземпляров этого процесса по причинам робастности или масштабирования, но в основном весь код упакован в один единственный процесс. На самом деле, как таковые, эти однопроцессные системы могут быть простыми распределенными системами, поскольку они почти всегда, в конечном счете, занимаются чтением данных из базы данных или их сохранением в ней.

Рис. 1.6. Однопроцессный монолит: весь код упакован в один единственный процесс

Эти однопроцессные монолиты, вероятно, представляют собой подавляющее большинство монолитных систем, с которыми по моему опыту люди воюют, и, следовательно, являются теми типами монолитов, на которых мы сосредоточим большую часть нашего времени. С этого момента, когда я буду использовать термин "монолит", я буду говорить именно о таких монолитах, если не скажу иначе.

И модульный монолит

Модульный монолит является вариацией как подмножество однопроцессного монолита: один процесс состоит из отдельных модулей, над каждым из которых можно работать независимо, но все они по-прежнему должны объединяться для развертывания, как показано на рис. 1.7. Концепция разбиения софта на модули не является чем-то новым, мы вернемся к некоторым историческим моментам вокруг нее позже в этой главе.

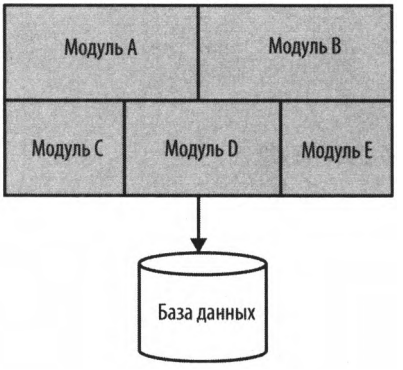


Рис. 1.7. Модульный монолит: код внутри процесса разбивается на модули

Для многих организаций модульный монолит — отличный вариант выбора. Если контуры модулей четко определены, то он обеспечивает высокую степень параллельности работы, но обходит трудности более распределенной архитектуры, основанной на микрослужбах, наряду с гораздо более простыми вопросами развертывания. Компания Shopify представляет собой отличный пример организации, которая использовала этот метод в качестве альтернативы декомпозиции на микрослужбы, и, похоже, для этой компании он действительно работает хорошо[5].

Одна из преград модульного монолита заключается в том, что база данных имеет тенденцию к нехватке декомпозиции, которую мы находим на уровне кода, приводя к значительным трудностям, с которыми можно столкнуться, если вы захотите затащить монолит в будущее. Я видел, как некоторые группы пытались продвинуть идею модульного монолита дальше, разложив базу данных по тем же линиям, что и модули, как показано на рис. 1.8. По существу дела, внести такое изменение в су-

[5] Выступление Кристен Уестейнд (Kirsten Westeinde) на YouTube (**http://bit.ly/2oauZ29**) дает глубокое понимание образа мыслей, принятого в Shopify, лежащего в основе привлечения модульного монолита вместо микрослужб.

ществующий монолит по-прежнему будет очень трудно, даже если вы оставите код в покое — многие шаблоны, которые мы разведаем в *главе 4*, помогут, если вы захотите попробовать сделать что-то подобное самостоятельно.

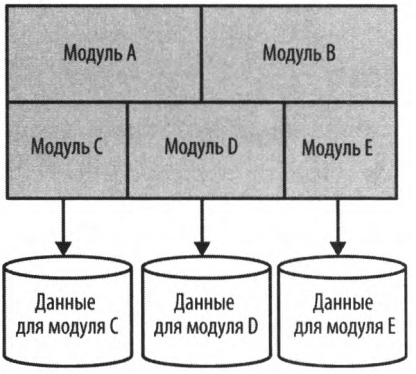


Рис. 1.8. Модульный монолит с разложенной базой данных

Распределенный монолит

> "Распределенная система — это система, в которой отказ компьютера, о существовании которого вы даже не подозревали, сделает ваш собственный компьютер непригодным для использования"[6].
>
> *– Лесли Лэмпорт (Leslie Lamport)*

Распределенный монолит — это система, состоящая из многочисленных служб, но по какой-то причине вся система целиком должна развертываться вместе. Распределенный монолит может вполне соответствовать определению архитектуры, ориентированной на службы (SOA), но слишком часто не выполняет ее обещания. По моему опыту, распределенные монолиты обладают всеми недостатками распределенной системы, а также недостатками однопроцессного монолита, не имея достаточного числа достоинств ни того, ни другого. Столкновение с распределенными монолитами в моей работе во многом повлияло на мой собственный интерес к архитектуре на основе микрослужб.

Распределенные монолиты обычно возникают в среде, где недостаточно внимания уделялось таким понятиям, как сокрытие информации и связность бизнес-функциональности, приводя вместо этого к высокосопряженным архитектурам, в которых изменения проникают через контуры служб, а кажущиеся невинными изменения, которые внешне выглядят локальными по охвату, разрушают другие части системы.

---

[6] Сообщение электронной почты, отправленное на доску объявлений DEC SRC в 12:23:29 тихоокеанского летнего времени (PDT) 28 мая 1987 года (подробности см. по адресу **https://www.microsoft.com/en-us/research/publication/distribution/**).

Сторонние черно-ящичные системы

В качестве монолитов, которые можно "разложить" в рамках усилий по миграции, можно также рассматривать некий сторонний софт. Сюда входят такие вещи, как платежные системы, CRM-системы и HR-системы. Общим фактором здесь является то, что этот софт разработан другими людьми, и у вас нет возможности изменить код. Указанный софт может быть серийным, развернутым на собственной инфраструктуре, либо используемым вами продуктом SaaS (программно-информационное обеспечение как служба). Многие методы декомпозиции, которые мы рассмотрим в этой книге, могут применяться даже в системах, в которых невозможно изменить лежащий в основании код.

Трудности монолитов

Монолит, будь то однопроцессный или распределенный, часто бывает уязвимее к опасностям сопряженности, в частности, имплементационной сопряженности и сопряженности развертывания, темы, которые вскоре мы разведаем подробнее.

По мере того как в одном и том же месте работает все больше и больше людей, они начинают вставать друг у друга на пути. Разные разработчики хотят изменить один и тот же фрагмент кода, разные группы хотят вывести функциональность в прямой эфир в разное время (или задержать развертывание). Возникает путаница вокруг того, кто чем владеет и кто принимает решения. Многочисленные исследования выявляют трудности, связанные с запутанными "линиями владения"[7]. Я называю эту проблему *конкуренцией за доставку* (delivery contention).

Наличие монолита не означает, что вы обязательно столкнетесь с проблемами конкуренции за доставку, так же как наличие архитектуры на основе микрослужб не означает, что вы никогда не столкнетесь с этой проблемой. Но архитектура на основе микрослужб дает вам более конкретные контуры в системе, вокруг которых могут быть проведены "линии владения", что дает вам гораздо больше гибкости в отношении того, как уменьшить эту проблему.

Преимущества монолитов

Однопроцессный монолит, однако, имеет целый ряд преимуществ. Его гораздо более простая топология развертывания позволяет избежать многих подводных камней, связанных с распределенными системами. Он приводит к гораздо более простым трудовым потокам по разработке софта. Мониторинг, устранение неполадок и такие мероприятия, как сквозное тестирование, также будут значительно упрощены.

---

[7] Microsoft Research провела исследования в этом пространстве, и я рекомендую их все. В качестве отправной точки я предлагаю статью "Не трогай мой код! Обследование влияния владения на качество программно-информационного обеспечения" (Don't Touch My Code! Examining the Effects of Ownership on Software Quality, http://bit.ly/2p5RIT1) Кристиана Берда (Christian Bird) и др.

Монолиты также упрощают многоразовое использование кода в самом монолите. Если мы хотим задействовать код в распределенной системе повторно, то мы должны решить, хотим ли мы копировать код, извлечь библиотеки или заложить совместную функциональность в службу. С монолитом наши варианты выбора намного проще, и многим людям нравится эта простота — весь код тут, просто пользуйся им!

К сожалению, люди стали рассматривать монолит как нечто, чего следует избегать, что по своей сути несет в себе проблемы. Я встречал многих, для которых термин "монолит" является синонимом унаследованной системы. И в этом проблема. Монолитная архитектура — это вариант, причем допустимый. Она не будет правильным вариантом выбора во всех обстоятельствах, так же как и микрослужбы, но, тем не менее, это вариант. Если мы попадем в ловушку систематического очернения монолита как жизнеспособного варианта для доставки нашего софта, то существует риск того, что мы сами либо наши пользователи поступим неправильно. В *главе 3* мы продолжим разведывание компромиссов между монолитами и микрослужбами и обсудим некоторые инструменты, которые помогут вам лучше оценить по достоинству то, что из существующего подходит для вашего собственного контекста.

О сопряженности и связности

Понимание балансирующих сил между сопряженностью и связностью имеет большое значение во время определения контуров микрослужб. Сопряженность (coupling) говорит о том, как изменение в одной вещи требует изменения в другой; связность (cohesion) говорит о том, как мы группируем родственный код. Эти понятия непосредственно связаны между собой.

Закон Константина хорошо формулирует эту связь[8]:

> "Структура является стабильной, если связность высокая, а сопряженность низкая".
>
> *– Ларри Константин (Larry Constantine)*

Это наблюдение кажется разумным и полезным. Если у нас есть два фрагмента родственного кода, то связность является низкой, поскольку родственная функциональность распространяется на каждый из них. У нас также есть тесная сопряженность, поскольку когда этот родственный код изменяется, то должны измениться обе вещи.

Если структура нашей системы кода меняется, то справляться с этим будет дорого, т. к. стоимость изменения контуров служб в распределенных системах очень высока. Если необходимо внести изменения в одну или несколько независимо разверты-

[8] Связность (cohesion, синонимы — когезия, спайка) — это показатель отношений внутри модуля и является внутримодульным понятием. Сопряженность (coupling, синонимы — сцепление, сцепка, стыковка) — это показатель отношений между модулями и является межмодульным понятием. См. **https://www.geeksforgeeks.org/software-engineering-differences-between-coupling-and-cohesion/** — *Пер.*

ваемых служб, возможно, улаживая влияние переломных изменений на контракты со службами, то такая необходимость, скорее всего, будет огромным тормозом.

Проблема с монолитом заключается в том, что слишком часто он является противоположностью того и другого. Вместо того чтобы тяготеть к связности и держать все вещи, которые тяготеют к одновременному изменению, вместе мы приобретаем и склеиваем всевозможный неродственный код. Схожим образом слабая сопряженность на самом деле не существует: если я хочу внести изменения в строку кода, то я могу сделать это достаточно легко, но я не смогу развернуть это изменение, не оказывая потенциального влияния на большую часть остального монолита, и мне, безусловно, придется разворачивать всю систему заново.

Мы также хотим стабильности системы, потому что наша цель, где это возможно, — воплотить в жизнь концепцию независимой развертываемости. Иными словами мы хотели бы иметь возможность вносить изменения в нашей службе и разворачивать эту службу в производство без необходимости менять что-либо еще. Для того чтобы это работало как надо, необходима стабильность служб, которые мы потребляем, и нам требуется обеспечить стабильный контракт тем службам, которые потребляют нас.

Учитывая массу информации об этих терминах, здесь с моей стороны было бы глупо слишком многое пересматривать, но я думаю, что резюме будет правомерным, в особенности для помещения этих идей в контекст архитектур на основе микрослужб. В конечном счете, понятия связности и сопряженности оказывают огромное влияние на то, как мы думаем об архитектуре, основанной микрослужбах. И это неудивительно — связность и сопряженность представляют собой вопросы, относимые к модульному софту, а что такое архитектура на основе микрослужб, как не модули, которые общаются через сети и которые могут быть развернуты независимо?

Краткая история сопряженности и связности

Понятия связности и сопряженности существуют в вычислительном процессе уже давно, причем эти понятия впервые были изложены Ларри Константином в 1968 году. Сдвоенные идеи сопряженности и связности легли в основу многих наших представлений о строительстве компьютерных программ. Такие книги, как "Структурный дизайн" (Structured Design) Ларри Константина и Эдварда Йордона (Edward Yourdon) (Prentice Hall, 1979) впоследствии повлияли на поколения программистов (для моей собственной университетской степени она была обязательным чтением, почти через 20 лет после ее первой публикации).

Ларри впервые изложил свои концепции связности и сопряженности в 1968 году (этот год был в особенности выдающимся для вычислительной техники) на Национальном симпозиуме по модульному программированию, той самой конференции, где закон Конвея впервые получил свое название. Этот год также дал нам две нашумевшие спонсируемые НАТО конференции, во время которых понятие инженерии программно-информационного обеспечения также получило свою известность (данный термин ранее был придуман Маргарет Х. Гамильтон).

Связность

Одно из самых лаконичных определений, в котором дается описание связности, из всех, что я слышал, таково: "код, который изменяется вместе, остается вместе". Для наших целей это определение является довольно хорошим. Как мы уже обсуждали, мы оптимизируем нашу архитектуру, основанную на микрослужбах, вокруг простоты внесения изменений в бизнес-функциональность, поэтому мы хотим, чтобы функциональность была сгруппирована так, чтобы мы могли вносить изменения как можно в меньшем количестве мест.

Если я хочу изменить руководство выпиской счетов-фактур, то мне не нужно выслеживать функциональность, которая нуждается в изменении в многочисленных службах, а затем координировать релиз этих только что измененных служб для внедрения нашей новой функциональности. Вместо этого я хочу обеспечить, чтобы изменение предусматривало модификации как можно меньшего числа служб, сохраняя стоимость изменений низкой.

Сопряженность

> "Сокрытие информации, как и диету, несколько легче описать, чем ее придерживаться".
>
> *–Дэвид Парнас (David Parnas),*
> Тайная история сокрытия информации (The Secret History Of Information Hiding)

Нам нравится связность, но мы остерегаемся сопряженности. Чем больше вещей "сопряжено", тем больше они должны меняться вместе. Но существуют разные типы сопряженности, и каждый из них потребует разных решений.

Что касается категоризации типов сопряженности, то существует много предыдущих технических решений, в частности работы Мейера, Йордона и Константина. Я представляю свою собственную, вовсе не говоря о том, что ранее проделанная работа ошибочна, просто я нахожу эту категоризацию полезнее, когда помогаю людям понять аспекты, ассоциированные с сопряженностью распределенных систем. Как таковая, она не претендует на исчерпывающую классификацию разных форм сопряженности.

Сокрытие информации

Концепция под названием "сокрытие информации" возвращается вновь и вновь в том, что касается дискуссий вокруг сопряженности. Указанная концепция, впервые изложенная Дэвидом Парнасом в 1971 году, появилась в его работе, посвященной определению границ модулей[9].

Стержневая идея скрытия информации заключается в том, чтобы отделять части кода, которые меняются часто, от тех, которые являются статическими. Мы хотим, чтобы

[9] Хотя в качестве источника часто цитируется известная работа Парнаса 1972 года "О критериях, используемых при декомпозиции систем на модули" (On the Criteria to be Used in Decomposing Systems into Modules), он впервые поделился этой концепцией в работе "Информационные аспекты методологии дизайна" (Information Distributions Aspects of Design Methodology), труды съезда IFIP '71, 1971.

граница модуля была стабильной, и она должна скрывать те части имплементации модуля, которые мы ожидаем изменять чаще. Идея заключается в том, что внутренние изменения могут вноситься безопасно до тех пор, пока поддерживается совместимость модуля.

Лично я принимаю подход, заключающийся в том, чтобы как можно меньше выставлять наружу за пределы границы модуля (или контура микрослужбы). Как только нечто становится частью интерфейса модуля, трудно вернуться назад. Но если вы спрячете это сейчас, то вы всегда сможете решить поделиться этим позже.

Инкапсуляция как понятие в объектно-ориентированном программно-информационном обеспечении является родственным термином, но в зависимости от того, чьего определения вы придерживаетесь, бывает не совсем тем же самым. Инкапсуляция в программировании стала означать упаковывание вместе одной или нескольких вещей в контейнер — подумайте о классе, содержащем как поля, так и методы, которые действуют на этих полях. Затем вы можете использовать видимость в определении класса для сокрытия части имплементации.

Для более детального знакомства с историей сокрытия информации я рекомендую книгу Парнаса "Тайная история сокрытия информации"[10].

Имплементационная сопряженость

Имплементационная сопряженность в типичных ситуациях является наиболее пагубной формой сопряженности, которую я встречаю, но, к счастью для нас, уменьшить ее нередко проще всего. При имплементационной сопряженности *A* сопряжено с *B* с точки зрения того, как имплементировано *B* — когда имплементация *B* изменяется, *A* изменяется тоже.

Трудность здесь в том, что детали имплементации часто произвольно выбраны разработчиками. Существует много способов решить некую проблему; мы выбираем один, но можем передумать. Когда мы решаем изменить свое мнение, мы не хотим, чтобы это ударило по потребителям (независимая развертываемость, помните?).

Классический и распространенный пример имплементационной сопряженности — использование совместной базы данных. На рис. 1.9 наша служба "Заказ" регистрирует все заказы, размещаемые в нашей системе. Служба "Рекомендации" предлагает нашим клиентам записи, которые они, возможно, захотят приобрести на основе предыдущих покупок. В настоящее время служба "Рекомендации" напрямую обращается к этим данным из базы данных.

"Рекомендации" требуют информации о том, какие заказы были размещены. В какой-то степени это представляет собой неизбежную доменную сопряженность, о которой мы поговорим чуть позже. Но в этой конкретной ситуации мы имеем сопряженность с определенной схемной структурой, диалектом SQL и, возможно, даже содержимым строк. Если служба "Заказ" изменяет имя столбца, разбивает таблицу "Клиентский заказ" или что-то еще, то она концептуально по-прежнему содержит информацию о заказе, но мы разрушаем то, как служба "Рекомендации"

---

[10] См. Парнас, Дэвид, "Тайная история сокрытия информации" (The Secret History of Information Hiding). Опубликовано в журнале Software Pioneer, ред. М. Брой и Э. Денерт (Berlin Heidelberg: Springer, 2002).

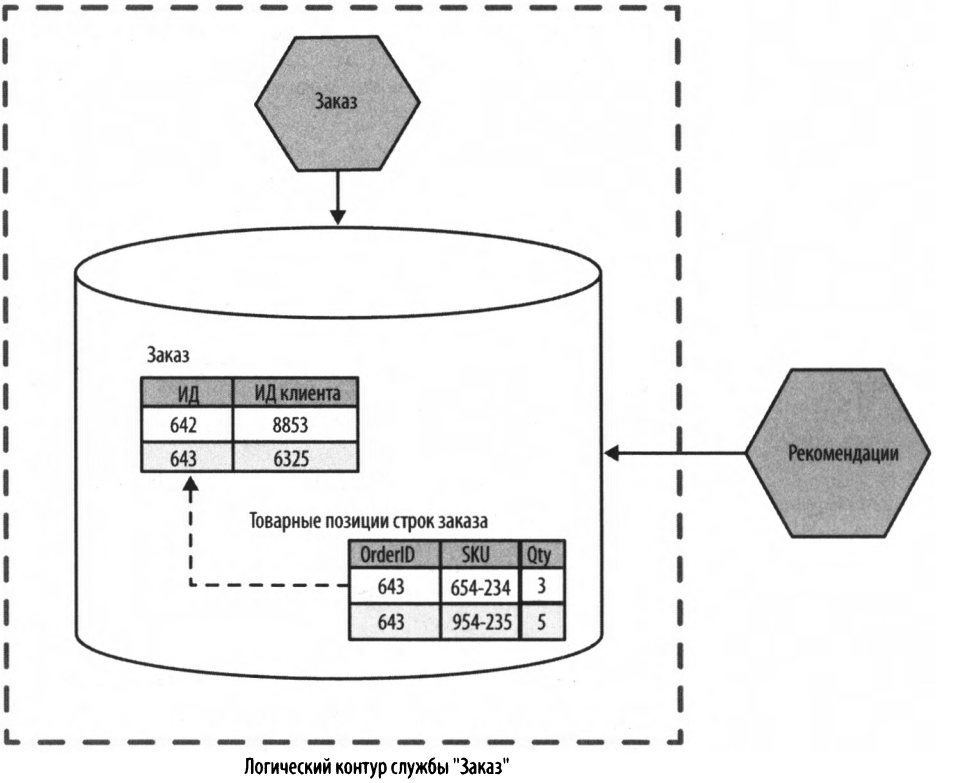


Рис. 1.9. Служба "Рекомендации" напрямую обращается к данным, хранящимся в службе "Заказ"

извлекает эту информацию. Лучше скрыть эту деталь имплементации, как показано на рис. 1.10, теперь служба "Рекомендации" получает доступ к необходимой информации через вызов API.

Мы также можем сделать так, чтобы служба "Заказ" публиковала набор данных в форме базы данных, которую предполагается использовать для массового доступа потребителей, как показано на рис. 1.11. До тех пор пока служба "Заказ" публикует данные соответствующим образом, любые изменения, вносимые внутрь службы "Заказ", невидимы для клиентов, поскольку она поддерживает публичный контракт. Она также открывает возможность улучшить модель данных, выставленную наружу для клиентов, подстраиваясь под их нужды. Мы разведаем подобные шаблоны подробнее в *главах 3* и *4*.

В обоих предыдущих примерах мы практически используем сокрытие информации. Акт сокрытия базы данных за четко определенным интерфейсом службы позволяет службе ограничить объем того, что выставляется наружу, и дает нам возможность изменять представление этих данных.

Еще одна полезная хитрость заключается в использовании мышления "извне-вовнутрь" в том, что касается определения интерфейса службы: управляй интерфейсом службы, сначала думая о вещах с точки зрения потребителей службы, а затем вырабатывай способ имплементации контракта с этой службой. Альтернатив-

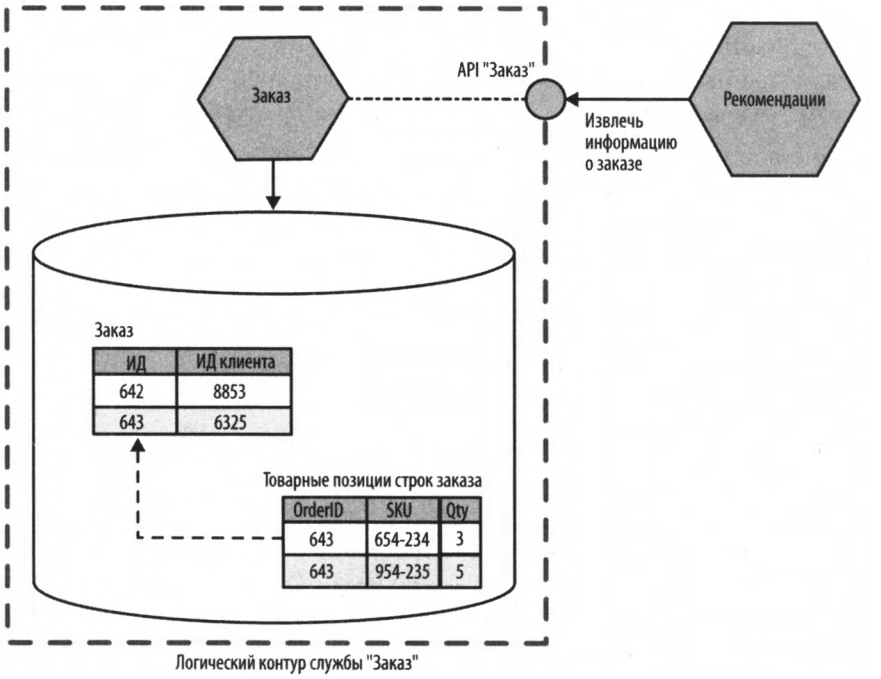


Рис. 1.10. Служба "Рекомендации" теперь обращается к информации о заказе через API, скрывая внутренние детали имплементации

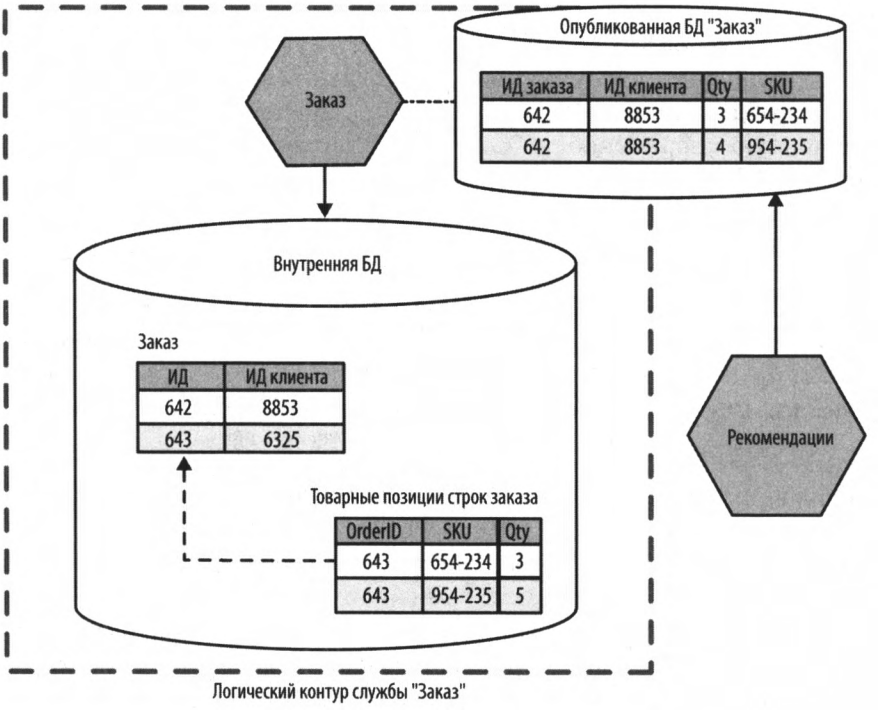


Рис. 1.11. Служба "Рекомендации" теперь обращается к информации о заказе через выставленную наружу базу данных, которая структурирована по-другому в отличие от внутренней базы данных

ный подход (который, по моим наблюдениям, к сожалению, слишком часто встречается) заключается в выполнении обратного. Группа, работающая над службой, берет модель данных или еще одну деталь внутренней имплементации, а затем думает о том, чтобы выставить ее наружу внешнему миру.

Думая "извне-вовнутрь", вы вместо этого сначала спрашиваете: "Что нужно потребителям моей службы?" И я не имею в виду, что вы спрашиваете *себя*, что нужно вашим потребителям; на самом деле я имею в виду, что вы спрашиваете у людей, которые будут вызывать вашу службу!

Рассматривайте интерфейсы, которые ваша микролужба выставляет наружу, как пользовательский интерфейс. Используйте мышление "извне-вовнутрь" для придания формы дизайну интерфейса в партнерстве с людьми, которые будут вызывать вашу службу.

Думайте о своем контракте между службой и внешним миром как о пользовательском интерфейсе. При разработке пользовательского интерфейса вы спрашиваете пользователей, чего они хотят, и итеративно проходите по дизайну интерфейса вместе со своими пользователями. Вы должны формировать свой контракт со службой таким же образом. В итоге вы получаете службу, которую вашим потребителям легче использовать, кроме того, это также помогает оставлять некоторое разграничение между внешним контрактом и внутренней имплементацией.

Временная сопряженность

Временная сопряженность — это в первую очередь вопрос времени выполнения, который в общем-то касается одной из ключевых трудностей синхронных вызовов в распределенной среде. Момент, когда сообщение отправляется, и то, как это сообщение обрабатывается, связаны во времени, причем принято говорить, что мы имеем временну́ю сопряженность. Это звучит немного странно, поэтому давайте рассмотрим явный пример на рис. 1.12.

Рис. 1.12. Можно сказать, что три службы, которые используют синхронные вызовы для выполнения операции, сопряжены во времени

Здесь мы видим синхронный HTTP-вызов, выполненный из нашей службы "Склад" в нижестоящую службу "Заказ" для получения требуемой информации о заказе. Для удовлетворения запроса служба "Заказ", в свою очередь, должна получить информацию от клиента, опять же через синхронный HTTP-вызов. Для завершения этой совокупной операции службы "Склад", "Заказ" и "Клиент" должны быть активными и доступными для контакта. Они сопряжены во времени.

Мы можем уменьшить эту проблему различными способами. Мы можем подумать о кэшировании: если служба "Заказ" кэширует информацию, необходимую ей от

службы "Клиент", то в некоторых случаях служба "Заказ" сможет избежать временной сопряженности с нижестоящей службой. Мы также можем подумать об асинхронном транспорте для отправки запросов, возможно, используя что-то вроде брокера сообщений. Это позволит отправлять сообщение в нижестоящую службу и обрабатывать его после того, как будет доступна нижестоящая служба.

Полное изложение коммуникации типа "служба-служба" выходит за рамки данной книги, но подробнее рассматривается в *главе 4* книги "Создание микросервисов".

Сопряженность развертывания

Возьмем отдельный процесс, состоящий из нескольких статически связанных модулей. Изменение вносится в одну строку кода в одном из модулей, и мы хотим развернуть это изменение. Для этого мы должны развернуть весь монолит — даже включая те модули, которые остаются неизменными. Все должно быть развернуто вместе, поэтому мы имеем сопряженность развертывания.

Сопряженность развертывания бывает принудительной, как в примере нашего статически связанного процесса, но и бывает вопросом выбора, обусловленным такими практиками, как релизная серия (release train). В случае релизной серии предварительно запланированные графики релизов составляются заранее, в типичной ситуации с периодическим графиком. Когда релиз готов к выпуску, все изменения, внесенные с момента последнего релиза в релизной серии, будут развернуты. Для некоторых людей релизная серия является полезным методом, но я решительно предпочитаю смотреть на нее не как на конечную цель, а как на переходный шаг к правильной методике релиза по требованию. Я даже работал в организациях, которые развертывали все службы в системе одновременно в рамках таких процессов релизных серий, не задумываясь о том, нужно ли менять эти службы.

Развертывание чего-либо несет в себе риск. Существует много путей снизить риск развертывания, и один из них — изменять только то, что должно быть изменено. Если мы сможем уменьшить число развертываний, возможно, путем декомпозиции более крупных процессов на независимо развертываемые микрослужбы, то мы сможем уменьшить риск каждого развертывания, сократив объем развертывания.

Меньшие релизы способствуют меньшему риску. Существует меньше того, что пойдет не так. Если все-таки что-то пошло не так, то выяснить то, что пошло не так и как это исправить, становится легче, потому что мы внесли меньше изменений. Поиск путей уменьшения размера релиза лежит в основе непрерывной доставки, которая поддерживает важность методов быстрой обратной связи и релиза по требованию[11]. Чем меньше объем релиза, тем проще и безопаснее его инициировать, и тем быстрее мы получим обратную связь. Мой собственный интерес к микрослужбам проистекает из предыдущей заинтересованности в непрерывной до-

[11] Более подробную информацию см. в Jez Humble (Джез Хамбл) и David Farley (Дэвид Фарли) "Непрерывная доставка: надежные релизы программно-информационного обеспечения посредством сборки, тестирования и автоматизации развертывания" (Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Upper Saddle River: Addison Wesley, 2010).

ставке — я искал архитектуры, которые облегчали бы адаптацию непрерывной доставки.

Уменьшение сопряженности развертывания, конечно же, не требует наличия микрослужб. Среды выполнения, такие, как Erlang, обеспечивают "горячее" развертывание новых версий модулей прямо в работающем процессе. В конце концов, пожалуй, многие из нас имеют доступ к таким возможностям в технологических стеках, которые мы используем ежедневно[12].

Доменная сопряженность

По существу дела, в системе, состоящей из нескольких независимых служб, должно быть некоторое взаимодействие между участниками. В архитектуре на основе микрослужб результатом является доменная сопряженность — взаимодействия между службами моделируют взаимодействия в нашей реальной области деятельности (реальном домене). Если вы хотите разместить заказ, то вам нужно знать, какие товарные позиции были в корзине покупок клиента. Если вы хотите отправить продукт, то вы должны знать, куда вы его отправляете. В нашей архитектуре на основе микрослужб эта информация по определению содержится в разных службах.

В качестве конкретного примера возьмем Music Corp. У нас есть склад, который хранит товары. Когда клиенты размещают заказы на компакт-диски, люди, работающие на складе, должны понимать, какие товарные позиции нужно взять и упаковать и куда нужно отправить пакет. Поэтому информация о заказе должна быть передана людям, работающим на складе.

На рис. 1.13 показан соответствующий пример: служба "Обработка заказа" отправляет все детали заказа в службу "Склад", которая затем запускает процесс упаковки номенклатурной товарной позиции. В рамках этой операции служба "Склад" использует ИД клиента для получения информации о клиенте из отдельной службы "Клиент", благодаря которой мы знаем, как уведомить его во время отправки заказа.

В этой ситуации мы делимся всем заказом со складом, что не имеет смысла, т. к. складу нужна информация только о том, что упаковать и куда отправить. Им не нужно знать, сколько товарная позиция стоит (если им нужно включить счет-фактуру в пакет, то это передается как предварительно отрисованный PDF-файл). У нас также были бы проблемы с информацией, доступ к которой мы должны контролировать, поскольку она распространяется слишком широко, например, если мы делимся полным заказом, то в итоге мы выставим наружу данные кредитной карты службам, которые в этом не нуждаются.

Поэтому вместо варианта, приведенного выше, мы можем предложить новое доменное понятие под названием "Инструкция по комплектации", содержащее только

---

[12] 10-е правило Гринспена (Greenspun) гласит: "Любая достаточно сложная программа на C или Fortran содержит нерегламентированную, неформально заданную, изобилующую дефектами, медленную имплементацию половины языка Common Lisp". Оно превратилось в новую шутку: "Каждая архитектура на основе микрослужб содержит наполовину сломанную реимплементацию на Erlang". Я думаю, что в этом есть много правды.

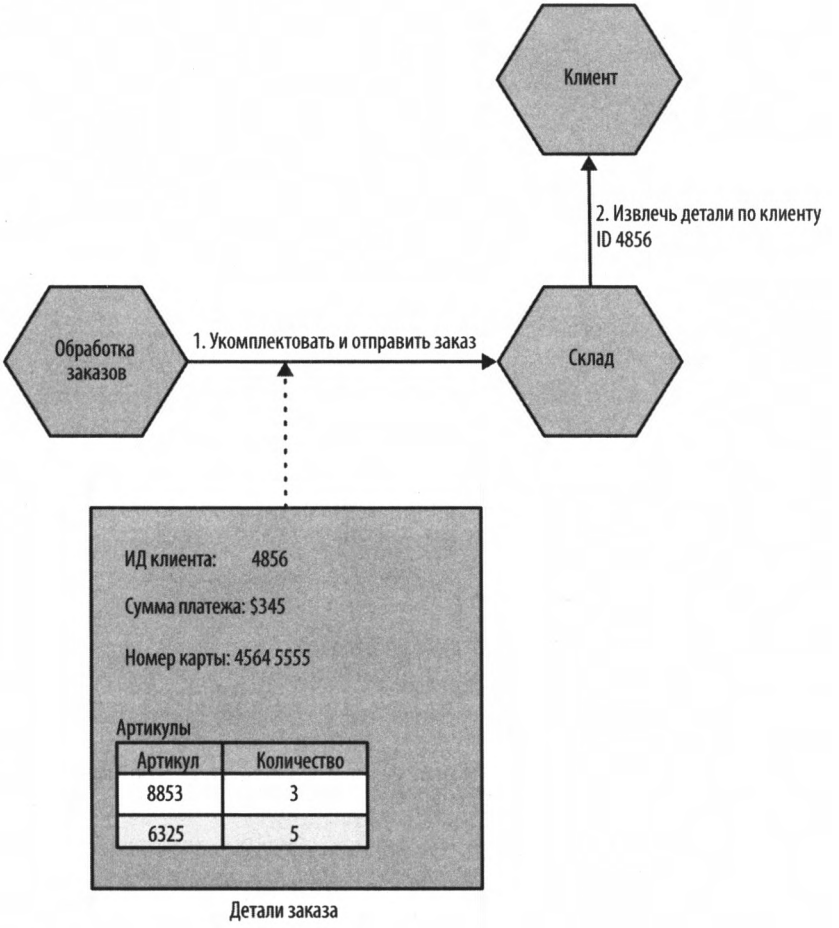


Рис. 1.13. Заказ отправляется на склад, для того чтобы начать его упаковку

ту информацию, которая необходима службе "Склад", как показано на рис. 1.14. Это еще один пример сокрытия информации.

Сопряженность можно уменьшить еще больше, удалив необходимость в том, чтобы служба "Склад" даже знала о клиенте, если мы этого захотим — вместо этого мы можем предоставить все надлежащие детали через "Инструкцию по комплектации", как показано на рис. 1.15.

Для того чтобы указанный подход работал как надо, это, вероятно, означает, что в какой-то момент "Обработка заказа" должна обратиться к службе "Клиент", чтобы иметь возможность сперва сгенерировать эту "Инструкцию по комплектации", но вполне вероятно, что "Обработка заказа" должна будет обратиться к информации о клиентах по другим причинам, так что это вряд ли будет большой проблемой. Из указанного процесса "отправки" "Инструкции по комплектации" следует вызов API из службы "Обработка заказа" в службу "Склад".

Альтернативой могло бы быть эмитирование "Обработкой заказа" некоего события, которое "Склад" потребляет, как показано на рис. 1.16. Путем эмитирования собы-

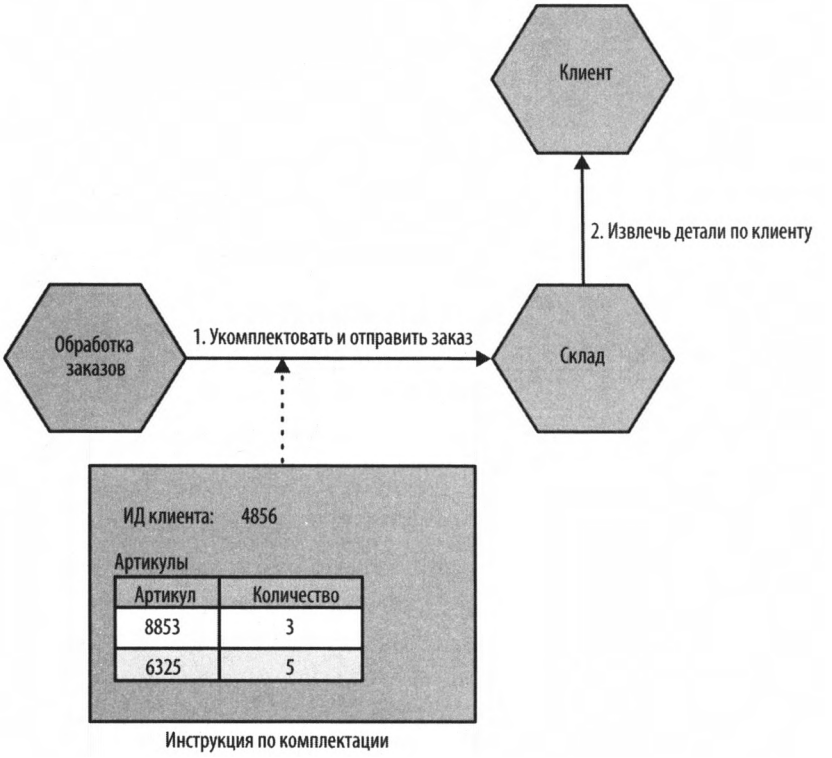


Рис. 1.14. Использование "Инструкции по комплектации" для уменьшения количества информации, которую мы отправляем в службу "Склад"

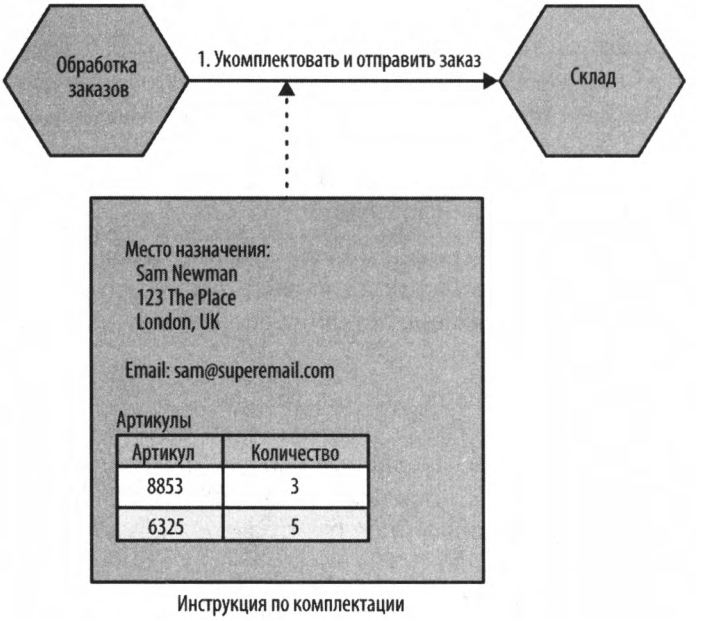


Рис. 1.15. Ввод дополнительной информации в "Инструкцию по комплектации" поможет избежать необходимости вызова службы "Клиент"

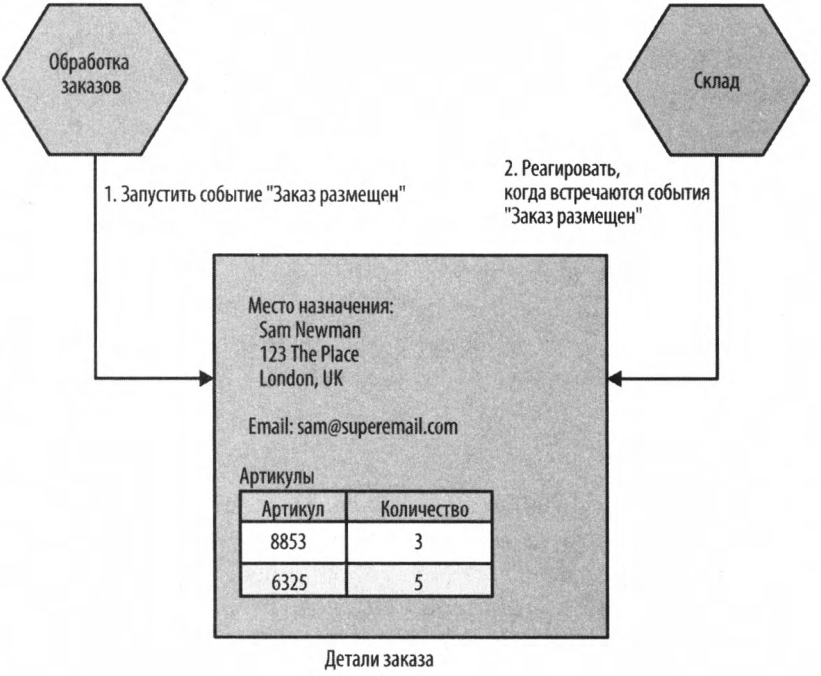


Рис. 1.16. Запуск события, которое способна получать служба "Склад", с информацией в объеме, достаточном для упаковки и отправки заказа

тия, которое "Склад" потребляет, мы фактически переворачиваем зависимости. Мы переходим от "Обработки заказа" в зависимости от службы "Склад", чтобы быть в состоянии обеспечить отправку заказа, к прослушиванию "Складом" событий из службы "Обработка заказа". Оба подхода имеют свои достоинства, и мой выбор, скорее всего, будет зависеть от более широкого понимания взаимодействий между логикой "Обработки заказа" и инкапсуляцией функциональности в службе "Склад" — это то, с чем помогает доменное моделирование, тема, которую мы разведаем далее.

По существу дела, для выполнения любой работы службой "Склад" все равно требуется некоторая информация о заказе. Мы не можем избежать этого уровня доменной сопряженности. Но, тщательно обдумывая то, какими понятиями мы делимся и как мы ими делимся, мы по-прежнему нацелены на снижение уровня используемой сопряженности.

Доменно-обусловленный дизайн

Как мы уже говорили, моделирование наших служб вокруг бизнес-домена имеет значительные преимущества для нашей архитектуры, основанной на микрослужбах. Вопрос в том, как сформулировать эту модель — и именно здесь выходит на сцену доменно-обусловленный дизайн (domain-driven design, DDD, дизайн, обусловленный областью деятельности).

Стремление к тому, чтобы наши программы лучше представляли реальный мир, в котором будут работать сами программы, не ново. Объектно-ориентированные языки программирования, такие, как Simula, были разработаны, для того чтобы дать нам возможность моделировать реальные области деятельности (реальные домены). Но чтобы эта идея действительно обрела форму, требуется нечто большее, чем возможности языка программирования.

В книге "Доменно-обусловленный дизайн" Эрика Эванса (Eric Evans)[13] изложен ряд важных идей, которые помогли нам лучше представлять проблемную область в наших программах. Полное исследование этих идей выходит за рамки данной книги, но я дам краткий обзор наиболее важных идей, возникающих при рассмотрении архитектур, основанных на микрослужбах.

Агрегат

В доменно-обусловленном дизайне агрегат представляет собой несколько запутанное понятие, имеющее массу разных определений. Не является ли он просто произвольной коллекцией объектов? Самой малой единицей, которую я должен взять из базы данных? Для меня всегда работала ментальная модель сначала рассматривать агрегат, взятый как представление понятия реальной области деятельности — подумайте о чем-то вроде "Заказа", "Счета-фактуры", "Номенклатурной товарной позиции" и т. д. В типичной ситуации с агрегатами связан жизненный цикл, что открывает их для имплементации в качестве "машины состояний". Мы хотим рассматривать агрегаты как самодостаточные единицы, мы хотим обеспечить, чтобы код, обрабатывающий переходы агрегата из состояния в состояние, группировался вместе с самим состоянием.

Размышляя об агрегатах и микрослужбах, можно сказать, что одна микрослужба будет обрабатывать жизненный цикл и хранение данных одного или нескольких разных типов агрегатов. Если функциональность в другой службе хочет изменить один из этих агрегатов, то она должна либо непосредственно запросить изменение в этом агрегате, либо заставить сам агрегат реагировать на другие вещи в системе с целью инициировать свои собственные переходы из состояния в состояние — примеры мы видим на рис. 1.17.

Здесь нужно понять главное — если внешняя сторона запрашивает в агрегате переход из одного состояния в другое, то агрегат может сказать "нет". В идеале вы хотите имплементировать свои агрегаты таким образом, чтобы незаконные переходы из состояния в состояние были невозможными.

Агрегаты могут иметь связи с другими агрегатами. На рис. 1.18 мы имеем агрегат "Клиент", который ассоциирован с одним или несколькими "Заказами". Мы решили моделировать "Клиента" и "Заказ" как отдельные агрегаты, которыми занимаются разные службы.

---

[13] Доменно-обусловленный дизайн: пути решения сложности в центре программно-информационного обеспечения (Domain-Driven Design: Tackling Complexity in the Heart of Software, Eric Evans, Boston: Addison-Wesley, 2004).

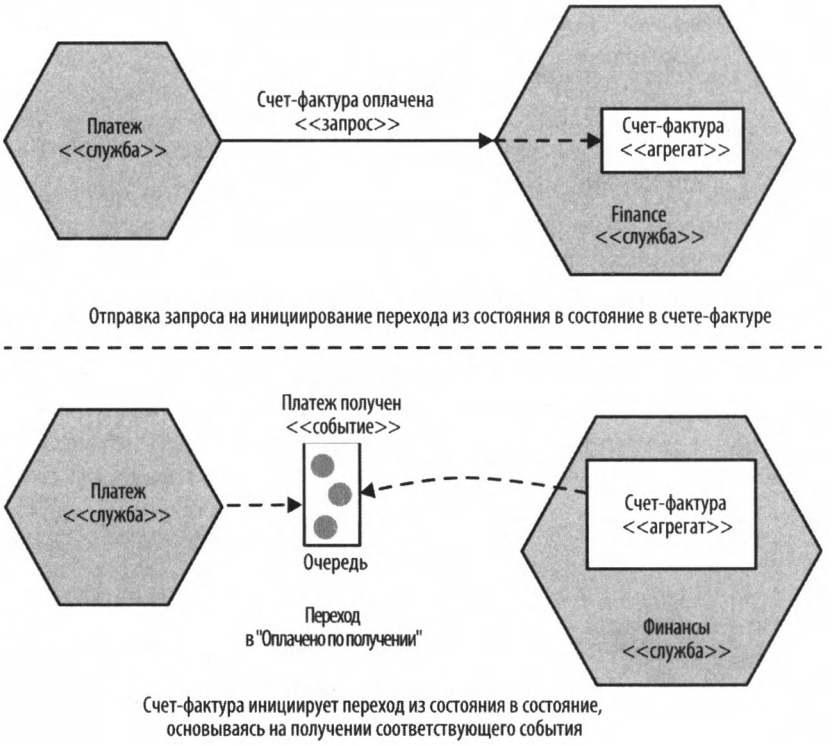


Рис. 1.17. Разные пути, которыми служба "Платеж" инициирует переход "Оплачено" в агрегате "Счет-фактура"

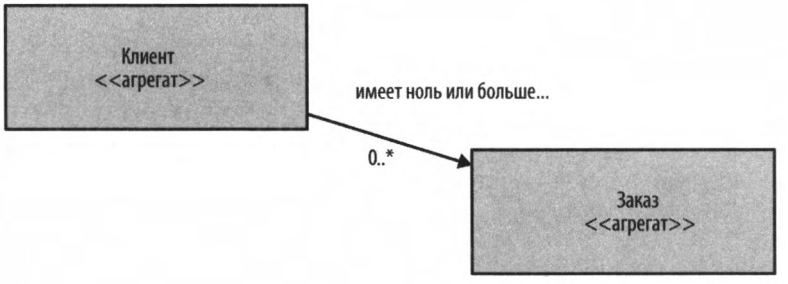


Рис. 1.18. Один агрегат "Клиент" может быть ассоциирован с одним или несколькими агрегатами "Заказ"

Существует масса способов разобрать систему на агрегаты, причем некоторые варианты весьма субъективны. Вы можете, по соображениям производительности или простоты имплементации, решить реформировать агрегаты через какое-то время. В самом начале, однако, я рассматриваю вопросы имплементации как второстепенные, исходно позволяя ментальной модели пользователей системы быть моим путеводным светом при первичном дизайне до тех пор, пока другие факторы не вступят в игру. В *главе 2* я представлю "Событийный штурм" как коллаборативное усилие в том, чтобы помочь сформировать эти доменные модели с помощью ваших коллег-неразработчиков.

Ограниченный контекст

Ограниченный контекст в типичной ситуации представляет собой более крупный организационный контур внутри организации. Внутри объема этого контура должны выполняться четкие обязанности. Это все немного запутано, поэтому давайте рассмотрим еще один конкретный пример.

В Music Corp наш склад представляет собой "кипучий муравейник": управление заказами, доставляемыми адресатам (и нерегулярным возвратом), поставка новых запасов, гонки на вилочных погрузчиках и т. д. В других местах финансовый отдел, возможно, менее веселый, но все же имеет важную функцию внутри нашей организации, обрабатывая платежную ведомость, оплачивая поставки и тому подобное.

Ограниченные контексты скрывают детали имплементации. Существуют внутренние вопросы, например, типы используемых вилочных погрузчиков мало интересуют кого-либо, кроме людей на складе. Эти внутренние вопросы должны быть скрыты от внешнего мира — им не нужно об этом знать, и они не должны об этом беспокоиться.

С точки зрения имплементации, ограниченные контексты содержат один или несколько агрегатов. Некоторые агрегаты могут быть выставлены наружу за пределы ограниченного контекста, другие могут быть скрыты внутри. Как и в случае агрегатов, ограниченные контексты могут иметь связи с другими ограниченными контекстами — при отображении в службы эти зависимости становятся зависимостями между службами.

Отображение агрегатов и ограниченных контекстов в микрослужбы

Как агрегат, так и ограниченный контекст дают нам единицы связности с четко определенными интерфейсами с более широкой системой. Агрегат — это самодостаточная машина состояний, которая фокусируется на единственном доменном понятии в нашей системе, а ограниченный контекст — это коллекция ассоциированных агрегатов, опять же с однозначно выраженным интерфейсом с более широким миром.

Поэтому и те, и другие могут хорошо работать в качестве контуров служб. Когда вы только начинаете, как я уже упоминал, я думаю, вы хотите уменьшить число служб, с которыми вы работаете. И, как результат, мне думается, вам следует, вероятно, ориентироваться на службы, которые охватывают все ограниченные контексты. Когда же вы встаете на ноги и решаете разложить эти службы на меньшие службы, старайтесь разделять их по контурам агрегатов.

Хитрость здесь заключается в том, что даже если вы решите позже разложить службу, моделирующую весь ограниченный контекст, на меньшие службы, то вы все равно скроете это решение от внешнего мира, возможно, представив потребителям более грубо гранулированный API. Решение выполнить декомпозицию службы на меньшие по гранулярности части, возможно, является имплементацион-

ным решением, поэтому с таким же успехом мы могли бы скрыть и его, если сможем!

Дальнейшее чтение

Тщательное изучение доменно-обусловленного дизайна будет полезно, но это выходит за рамки данной книги. Если вы хотите следовать ему дальше, то я предлагаю прочитать либо оригинальную книгу "Доменно-обусловленный дизайн" Эрика Эванса либо "Основы доменно-обусловленного дизайна" Вона Вернона[14].

Резюме

Как мы обсудили в этой главе, микрослужбы представляют собой независимо развертываемые службы, моделируемые вокруг бизнес-домена. Они общаются друг с другом через сеть. Мы используем принципы сокрытия информации вместе с доменно-обусловленным дизайном для создания служб со стабильными контурами, над которыми легче работать независимо, и мы делаем все возможное, чтобы уменьшить многие формы сопряженности.

Мы кратко описали историю того, откуда они возникли, и даже нашли время на то, чтобы взглянуть на небольшую часть огромного объема предыдущей работы, на которой они основываются. Мы также кратко рассмотрели некоторые трудности, ассоциированные с архитектурами на основе микрослужб. Эту тему я более подробно изложу в следующей главе, где мы также обсудим вопрос планирования транзита на архитектуру, основанную на микрослужбах, а также дадим рекомендации, которые помогут решить, подходят ли они вам вообще.

[14] См. Основы доменно-обусловленного дизайна (Domain-Driven Design Distilled, Vaughn Vernon, Boston: Addison-Wesley, 2014).

Планирование миграции

Слишком легко погрузиться сразу в мельчайшие технические стороны декомпозиции монолита — и это будет в центре внимания остальной части книги! Но сначала нам действительно нужно разведать несколько менее технических вопросов. С чего начинать миграцию? Как управлять изменениями? Как брать с собой других в это путешествие? И важный вопрос, который необходимо задать на ранней стадии, — использовать ли микрослужбы вообще?

Понимание цели

Микрослужбы не являются самоцелью. Вы не "выигрываете", если имеете микрослужбы. Решение внедрить архитектуру на основе микрослужб должно быть осознанным, основанным на принятии рациональных решений. Рассуждать о миграции на архитектуру, основанную на микрослужбах, следует для достижения чего-то, чего вы в настоящее время не способны достичь с помощью архитектуры существующей системы.

Если вы не будете понимать то, чего вы пытаетесь достичь, то каким образом вы собираетесь информировать свой процесс принятия решений о том, какие варианты вы должны принять? Внедряя микрослужбы, вы пытаетесь достичь состояния, которое значительно изменит то, на чем вы сосредоточиваете свое время и как вы расставляете приоритеты своих усилий.

Понимание цели также поможет вам не стать жертвой "аналитического паралича" — быть подавленным вариантами выбора. Вы также рискуете впасть в каргокультную ментальность, просто исходя из того, что "если микрослужбы хороши для Netflix, то они хороши и для нас!".

Часто встречающееся непонимание

Несколько лет назад я вел семинар по микрослужбам на конференции. Как и на всех моих занятиях, мне нравится вникать в то, почему люди туда приходят и что они надеются получить от семинара. На этом конкретном занятии несколько человек пришли из одной компании, и мне было любопытно узнать, почему компания их послала.

— В чем причина вашего посещения семинара? Почему у вас возник интерес к использованию микрослужб? — спросил я одного из них. И ответ был:

— Мы не знаем, наш босс велел нам прийти! — Заинтригованный, я продолжил расследование.

— Так, а как вы предполагаете, почему ваш босс захотел, чтобы вы были здесь?

— Ну, это вы у него спросите — он сидит позади нас, — ответил посетитель.

Я переключил свою серию вопросов на их босса, задав тот же вопрос:

— Итак, почему вы хотите использовать микрослужбы? — Ответ босса был таким:

— Наш технический директор сказал, что мы принимаемся за микрослужбы, поэтому я подумал, что мы должны узнать, что это такое!

Эта правдивая история, хотя с одной стороны и смешная, к сожалению, встречается слишком часто. Я сталкиваюсь со многими группами, которые приняли решение внедрить архитектуру на основе микрослужб, не понимая, почему или чего они надеются достичь.

Проблемы с отсутствием четкого видения относительно того, почему вы используете микрослужбы, носят разнообразный характер. Могут потребоваться значительные инвестиции непосредственно с точки зрения увеличения числа людей или количества денег, либо с точки зрения "приоритизации"[1] работы по транзиту, помимо добавления функций. Это осложняется еще более тем фактом, что потребуется некоторое время на то, чтобы увидеть выгоды от такого транзита. Иногда это приводит к ситуации, когда люди год или больше специализируются на транзите, но вообще не могут вспомнить, почему они его начали. Это не просто вопрос эффекта понесенных расходов, сотрудники буквально не знают, зачем они делают эту работу.

В то же время люди просят меня поделиться информацией о возвратности инвестиций (ROI) в переход на архитектуру, основанную на микрослужбах. Некоторые хотят иметь твердые факты и цифры, подтверждающие причину, почему они вообще должны рассматривать этот подход. Реальность такова, что детальные исследования такого рода вещей исчисляются единицами и встречаются редко, даже когда они существуют, наблюдения из такого рода исследований редко годятся для переноса на свою почву из-за разных контекстов, в которых люди могут оказаться.

И что же нам остается, угадайте? Ну, уж нет. Убежден, что мы можем и должны иметь более широкие исследования эффективности наших решений в области разработки, технологий и архитектуры. С этой целью уже проводится некоторая работа, например, в виде таких вещей, как "Отчет о состоянии дел в области DevOps" (The State of DevOps Report, **http://bit.ly/2ojVq5o**)[2], но там архитектура исследуется лишь мимоходом. Вместо этих скрупулезных исследований мы должны, по крайней мере, стремиться применять к нашему принятию решений критическое мышление и в то же время осваивать более экспериментальный настрой мыслей.

Вам потребуется четкое понимание того, чего вы надеетесь достичь. Ни один расчет показателя ROI не будет выполнен без правильной оценки того, какой возврат от инвестиций вы ищете. Нам нужно сосредоточиться на результатах, которых мы надеемся достичь, а не рабски и догматически придерживаться единого подхода.

---

[1] Здесь и далее оставлен перевод авторского термина — *Ред.*

[2] DevOps (разработка и операции) — это тесная связь между разработчиками приложений и людьми, которые их тестируют и развертывают. Считается, что DevOps является пересечением разработки программно-информационного обеспечения, проведения контроля его качества и его эксплуатации и сопровождения. Она призвана улучшить сотрудничество между этими группами.
См. **https://encyclopedia2.thefreedictionary.com/devops** — *Пер.*

Нам нужно ясно и разумно подумать о том, как лучше всего получить то, чего мы хотим, даже если это означает откладывание в сторону большой работы или возврат к старому доброму вышедшему из моды скучному подходу.

Три ключевых вопроса

Во время работы с организацией, чтобы помочь менеджменту понять, следует ли им подумать о внедрении архитектуры на основе микрослужб, я обычно задаю один и тот же набор вопросов:

Чего вы надеетесь достичь?

Это должен быть набор результатов, выровненных по целям, которые бизнес пытается достичь, и может быть артикулирован таким образом, который описывает выгоду для конечных пользователей системы.

Подумали ли вы об альтернативах использованию микрослужб?

Как мы увидим далее, часто существует масса других способов достижения некоторых из тех же выгод, которые приносят микрослужбы. Анализировали ли вы эти вещи? Если нет, то почему? Довольно часто вы получите то, что вам нужно, используя гораздо более простой и "скучный" метод.

Как вы узнаете, что транзит работает?

Если вы решите начать этот транзит, как вы узнаете, что вы идете в правильном направлении? Мы вернемся к этой теме в конце главы.

Не раз я обнаруживал, что, задавая эти вопросы, менеджмент компаний снова задумывается над тем, стоит ли им идти дальше с архитектурой на основе микрослужб.

Почему вы, возможно, выберете микрослужбы?

Я не могу определить цели, которые существуют в вашей компании. Вы лучше знаете цели своей компании и стоящие перед ней трудности. То, что я могу изложить, — это причины, которые часто упоминают, говоря о внедрении микрослужб компаниями по всему миру. В духе честности я также опишу другие способы, которыми вы могли бы потенциально достичь тех же результатов, используя другие подходы.

Повысить автономию групп

> "В какой бы индустрии вы ни работали, все дело в ваших людях, в том, чтобы ловить их на правильных поступках и обеспечивать им уверенность, мотивацию, свободу и желание достичь своего истинного потенциала".
>
> *– Джон Тимпсон (John Timpson)*

Многие организации продемонстрировали выгоды от создания автономных групп. Поддержание малого размера организационных групп, разрешение им создавать

тесные связи и эффективно работать вместе, не привлекая слишком много бюрократии, помогло многим организациям в росте и масштабировании эффективнее, чем у некоторых из их коллег. Гор (Gore) добился большого успеха, ограничив численность работающих в каждой структурной единице пределом в 150 человек, ради того чтобы все друг друга знали. Для того чтобы эти меньшие структурные единицы могли функционировать, им необходимо предоставлять полномочия и ответственность для работы в качестве автономных единиц.

Очень успешный британский ритейлер Timpsons добился огромных масштабов, расширяя полномочия своей рабочей силы, уменьшая потребность в центральных функциях и позволяя местным магазинам принимать решения самим, например, предоставляя им полномочия в том, сколько возвращать недовольным клиентам. Теперь председатель компании Джон Тимпсон (John Timpson) прославился тем, что отказался от внутренних правил и заменил их всего двумя:

Выгляди, как подобает, и можешь класть деньги в кассу.

Ты можешь сделать что-то еще ради более качественного обслуживания клиентов.

Автономия также работает и в меньшем масштабе, и большинство современных организаций, с которыми я сотрудничаю, стремятся создавать в своих организациях более автономные группы, часто пытаясь копировать модели из других организаций, таких, как модель двухпиццевых групп Amazon или модель Spotify[3].

Если все сделано правильно, то автономия группы расширит полномочия людей, поможет им подняться и вырасти, а также ускорит работу. Когда группы владеют микрослужбами и имеют полный контроль над этими службами, они увеличивают степень своей автономии в рамках более крупной организации.

Как еще это сделать?

Автономия — распределение ответственности — проявляется по-разному. Выяснение того, как заложить больше ответственности в группу, не требует сдвига в архитектуре. По сути, это процесс выявления конкретных обязанностей, которые могут быть переданы группам, и он сыграет свою роль во многих отношениях. Давая владеть частью кодовой базы разным группам, вы получите один из ответов (модульный монолит здесь по-прежнему принесет вам выгоду): это можно сделать путем выявления людей, уполномоченных принимать решения в отношении частей кодовой базы по функциональным основаниям (например, Райан умеет лучше показывать рекламу, поэтому он несет ответственность за это; Джейн знает больше всего о тонкой настройке производительности наших запросов, поэтому сначала пропускайте все в этой области через нее).

Улучшение автономии также играет свою роль в том, что просто не приходится больше ждать, пока другие люди сделают что-то для вас, поэтому принятие самообслуживаемых подходов к обеспечению работы машин или сред является огром-

[3] Которую, как известно, даже Spotify больше не использует.

ным стимулом, избегая необходимости в привлечении центральных операционных групп, которые выставляют наряды-заказы на ежедневные мероприятия.

Сократить время до рынка

Имея возможность вносить и развертывать изменения в отдельные микрослужбы, а также развертывать эти изменения, не дожидаясь согласованных релизов, мы можем выпускать функциональность для наших клиентов быстрее. Способность привлекать больше людей к решению проблемы также является фактором — вскоре мы это рассмотрим.

Как еще это сделать?

И с чего же начать? В игру вступает так много переменных, когда рассматриваешь вопрос о том, как быстрее поставлять софт. Я всегда предлагаю вам выполнить какое-то упражнение по моделированию пути к производству, т. к. оно помогает продемонстрировать, что самым большим блокирующим игроком может оказаться совсем не то, что вы думаете.

Помню, как много лет назад в одном крупном инвестиционном банке нас привлекли для того, чтобы помочь ускорить доставку софта. Нам сказали: "Разработчикам требуется слишком много времени на то, чтобы запускать все в производство!". Один из моих коллег, Киф Моррис (Kief Morris), нашел время, чтобы составить карту всех этапов, связанных с доставкой софта, рассматривая процесс с момента, когда владелец продукта придумывает идею, до момента, когда эта идея фактически попадает в производство.

Он быстро выяснил, что с того времени, когда разработчик приступал к выполнению части работы, до ее развертывания в производственной среде уходило в среднем около шести недель. Мы чувствовали, что могли бы скинуть с этого процесса пару недель за счет использования подходящей автоматизации, поскольку были задействованы ручные процессы. Но Киф обнаружил гораздо более серьезную проблему на пути к производству — часто требовалось более 40 недель на то, чтобы идеи от владельца продукта доходили до точки, где разработчики могли хоть как-то начать работу. Сосредоточив наши усилия на улучшении этой части процесса, мы помогли клиенту в уменьшении времени выхода на рынок новой функциональности значительно больше.

Поэтому подумайте обо всех шагах, связанных с доставкой софта. Посмотрите, сколько времени они занимают, продолжительность (как прошедшее время, так и занятое время) по каждому шагу и выделите "болевые" точки на этом пути. После всего этого вы вполне можете обнаружить, что микрослужбы будут частью решения, но вы, вероятно, найдете много другого, что можно попробовать параллельно.

Выполнить эффективное по стоимости масштабирование с учетом нагрузки

Разбив нашу обработку на отдельные микрослужбы, мы получаем возможность масштабировать эти процессы независимо. Это означает, что мы также можем на-

деяться на проведение эффективного по стоимости масштабирования. Нам нужно выполнить масштабирование вверх только тех частей нашей обработки, которые в настоящее время ограничивают нашу способность справляться с нагрузкой. Из этого также следует, что потом мы можем выполнить масштабирование вниз тех микрослужб, которые находятся под меньшей нагрузкой, возможно, даже выключая их, когда они не требуются. Отчасти именно поэтому так много компаний, создающих продукты SaaS, используют архитектуру на основе микрослужб — она дает им больше контроля над операционными расходами.

Как еще это сделать?

Здесь мы можем рассмотреть огромное число альтернатив, с большинством из которых легче проводить эксперименты, прежде чем связывать себя подходом, ориентированным на микрослужбы. Для начала мы могли бы просто взять "коробку" побольше — если вы находитесь в публичном облаке или другом типе виртуальных платформ, то вы могли бы обеспечить, чтобы в вашем процессе работали более крупные машины. Это "вертикальное" масштабирование, очевидно, имеет свои ограничения, но для быстрого краткосрочного улучшения его не следует сразу отбрасывать.

Традиционное горизонтальное масштабирование существующего монолита — главным образом с использованием нескольких копий — могло бы быть весьма эффективным. Выполнение нескольких копий вашего монолита на фоне механизма распределения нагрузки, такого, как балансировщик нагрузки или очередь, способно обеспечить простую обработку большей нагрузки, хотя это, пожалуй, не поможет, если узкое место находится в базе данных, которая, в зависимости от технологии, не поддерживает такой механизм масштабирования. Горизонтальное масштабирование легко можно попробовать, и вы действительно должны дать ему шанс, прежде чем рассматривать микрослужбы, поскольку оно, вероятно, быстро даст возможность оценить его пригодность и имеет гораздо меньше недостатков, чем полноценная архитектура на основе микрослужб.

Вы также могли бы поменять технологию, используемую с альтернативами, способную лучше справляться с нагрузкой. Однако это мероприятие в типичной ситуации является нетривиальным — возьмите работу по переносу существующей программы на новый тип базы данных или язык программирования. Сдвиг к микрослужбам, по сути дела, облегчит смену технологии, поскольку вы имеете возможность изменять технологию, используемую внутри только требующих ее микрослужб, держа остальные нетронутыми и тем самым уменьшая влияние замены.

Повысить робастность

Переход от однопользовательского софта к мультитенантным SaaS-приложениям означает, что влияние перебоев в работе системы будет гораздо более распространенным. Ожидания наших клиентов в отношении доступности их софта, а также его важность в их жизни, возросли. Раскладывая наше приложение на отдельные,

независимо развертываемые процессы, мы открываем массу механизмов повышения робастности наших приложений.

Используя микрослужбы, мы имплементируем более робастную архитектуру, поскольку функциональность раскладывается на части, т. е. влияние на одну область функциональности не обязательно разрушает всю систему в целом. Мы также можем сосредоточить наше время и энергию на тех частях приложения, которые требуют робастности больше всего, обеспечивая, чтобы критически важные части нашей системы оставались в рабочем состоянии.

Отказоустойчивость против робастности

Обычно, когда мы хотим улучшить способность системы избегать перебоев в работе, плавно обрабатывать сбои, когда они происходят, и быстро ее восстанавливать, когда возникают проблемы, мы часто говорим об отказоустойчивости. В области того, что сейчас известно как инженерия отказоустойчивости была проделана большая работа, которая рассматривала данную тему в целом, поскольку она применима ко всем областям, а не только к вычислительной технике. Модель отказоустойчивости, впервые предложенная Дэвидом Вудсом (David Woods), рассматривает концепцию отказоустойчивости шире и указывает на тот факт, что быть отказоустойчивым не так просто, как мы могли бы подумать, разграничивая нашу способность иметь дело с известными и неизвестными источниками сбоев среди прочего[4].

Джон Оллспоу (John Allspaw), коллега Дэвида Вудса, помогает различать понятия робастности и отказоустойчивости. Робастность — это способность иметь систему, которая может реагировать на ожидаемые вариации. Отказоустойчивость — это наличие организации, способной адаптироваться к тому, что не было продумано заранее, вполне включая в себя создание культуры экспериментирования посредством таких вещей, как инженерия хаоса. Например, мы отдаем себе отчет в том, что конкретная машина может заглохнуть, поэтому мы вносим в нашу систему избыточность путем балансировки экземпляра по нагрузке. Это пример обращения с робастностью. Отказоустойчивость есть процесс подготовки организации самой себя к тому факту, что она не в состоянии предвосхитить все потенциальные проблемы.

Важным соображением здесь является то, что микрослужбы не обязательно дают вам робастность бесплатно. Скорее, они открывают возможности для дизайна системы таким образом, чтобы она была терпимее к сетевым разделениям, перебоям в работе служб и т. п. Простое распределение функциональности между несколькими отдельными процессами и отдельными машинами не гарантирует улучшенной робастности; скорее наоборот — это просто увеличит "суммарную площадь" сбоя.

Как еще это сделать?

Выполняя несколько копий вашего монолита, возможно, на фоне балансировщика нагрузки или другого механизма распределения нагрузки, такого, как очередь[5], мы

---

[4] См. Дэвид Вудс, "Четыре концепции отказоустойчивости и последствия для будущего инженерии отказоустойчивости" (David Woods, “Four Concepts for Resilience and the Implications for the Future of Resilience Engineering.” Reliability Engineering & System Safety 141 (2015) 5–9).

[5] См. шаблон конкурирующего потребителя для одного такого примера в книге "Шаблоны интеграции предприятия" Грегора Хоп и Бобби Вулфа, стр. 502 (Enterprise Integration Patterns by Gregor Hohpe, Bobby Woolf).

добавим в нашу систему избыточность. Мы еще больше повысим робастность наших приложений, распределяя экземпляры монолита по нескольким плоскостям сбоя (например, не все машины находятся в одной стойке или одном центре обработки данных).

Инвестиции в более надежное аппаратное и программно-информационное обеспечение также принесут выгоду, как и тщательное исследование существующих причин перебоев в работе системы. Например, я видел немало производственных проблем, вызванных чрезмерной зависимостью от ручных процессов, или людей, которые "не следуют протоколу". Это часто означает, что невинная ошибка человека может иметь значительные последствия. Авиакомпания British Airways пережила массовый отток в 2017 году, в результате чего все ее рейсы в лондонский Хитроу и Гатвик были отменены. Эта проблема, по всей видимости, была непреднамеренно вызвана скачком напряжения в результате действий одного человека. Если робастность вашего приложения опирается на допущение, что люди никогда не ошибаются, то вы движетесь по ухабистой дороге.

Промасштабировать число разработчиков

Мы все, вероятно, встречались с проблемой, когда проект "закидывали" разработчиками в попытках его ускорить — это очень часто приводит к обратному эффекту. Но для решения некоторых проблем все-таки нужно больше людей. Как отмечает Фредерик Брукс (Frederick Brooks)[6] в своей ставшей теперь основополагающей книге "Мифический человеко-месяц", добавление большего числа людей продолжит улучшать быстроту осуществления доставки, если сама работа будет подразделена на отдельные части с ограниченным взаимодействием между ними. Он приводит пример уборки урожая в поле — простой задачи, в которой несколько человек работают параллельно, т. к. работа, выполняемая каждым сборщиком, не требует взаимодействия с другими людьми. Софт редко работает подобным образом, поскольку выполняемая работа не всегда одинакова, и часто выход из одной части работы необходим в качестве входа для другой.

Имея четко определенные контуры и архитектуру, которая сосредоточена вокруг того, чтобы наши микрослужбы ограничивали свою сопряженность друг с другом, мы придумываем фрагменты кода, над которыми можно работать независимо. Следовательно, мы надеемся, что сможем промасштабировать число разработчиков, сократив конкуренцию за доставку.

Для успешного масштабирования числа разработчиков, которых вы привлекаете к решению задачи, требуется хорошая степень автономии между самими группами. Просто наличия микрослужб будет недостаточно. Вам придется подумать о том, как группы выровнены по владельцу службы и какая требуется координация между группами. Вам также необходимо будет подразделить работу так, чтобы изменения не нужно было координировать по слишком большому числу служб.

---

[6] См. Фредерик Брукс "Мифический человеко-месяц" (Frederick P. Brooks, The Mythical Man-Month, 20th Anniversary Edition, Boston: Addison-Wesley, 1995).

Как еще это сделать?

Микрослужбы хорошо работают для более крупных групп, т. к. сами микрослужбы становятся размежеванными функциональными частями, над которыми можно работать независимо. Альтернативный подход состоял бы в том, чтобы подумать об имплементации модульного монолита: каждый модуль принадлежит разным группам, и, до тех пор пока интерфейс с другими модулями остается неизменным, они могут продолжать вносить изменения изолированно.

Однако этот подход несколько ограничен. У нас все еще есть некоторая форма конкуренции между разными группами, поскольку софт по-прежнему упаковывается вместе, поэтому акт развертывания по-прежнему требует координации между соответствующими участвующими сторонами.

Внедрить новую технологию

Монолиты, как правило, ограничивают наш выбор технологий. У нас обычно есть один язык программирования на бэкэнде, использующий одну идиому программирования. Мы привязаны к одной платформе развертывания, одной операционной системе, одному типу базы данных. С архитектурой на основе микрослужб мы получаем возможность разнообразить эти варианты для каждой службы.

Изолировав смену технологии в контуре одной службы, мы поймем выгоды новой технологии в изоляции и ограничим ее влияние, если окажется, что она имеет проблемы.

По моему опыту, в то время как ориентированные на микрослужбы зрелые организации часто ограничивают число поддерживаемых ими стеков технологий, они редко придерживаются однородного подхода в использовании ими технологий. Гибкость в способности апробировать новые технологии безопасным способом дает им конкурентное преимущество, как с точки зрения обеспечения более качественных результатов для клиентов, так и в плане поддержания удовлетворенности своих разработчиков, когда они приступают к освоению новых навыков.

Как еще это сделать?

Если мы по-прежнему продолжим поставлять наш софт как единый процесс, у нас действительно будут ограничения на то, какие технологии мы можем использовать. Конечно, мы могли бы безопасно внедрить новые языки в одной и той же среде выполнения — в качестве одного примера JVM способна с радостью разместить код, написанный на нескольких языках, в одном и том же работающем процессе. Однако новые типы баз данных становятся все более проблемными, поскольку из этого вытекает некоторая декомпозиция ранее монолитной модели данных для обеспечения поступательной (инкрементной) миграции, если только вы не собираетесь полностью и немедленно переключиться на новую технологию баз данных, что сделать сложно и рискованно.

Если текущий стек технологий считается "выгорающей платформой", то у вас не будет иного выбора, кроме как заменить его на новый, лучше поддерживаемый

стек технологий[7]. Конечно же, вам ничто не мешает постепенно поменять ваш существующий монолит на новый — шаблоны, наподобие подхода на основе фикусаудавки, описанного в *главе 3*, хорошо для этого работают.

Многоразовое использование?

Многоразовое использование — одна из наиболее часто заявляемых целей миграции на микрослужбы, и, по-моему, это слабая цель. По сути, многоразовое использование — это не тот прямой результат, который люди хотят. Люди надеются, что многоразовое использование приведет к другим выгодам. Мы надеемся, что благодаря ему мы сможем поставлять функции быстрее или, возможно, снизить затраты, но если эти вещи являются вашей целью, то просто отслеживайте их, а иначе вы в итоге будете оптимизировать совсем не то.

Поясню, что я имею в виду. Давайте повнимательнее приглядимся к одной из обычных причин, по которой многоразовое использование выбирается в качестве цели. Мы хотим поставлять функции быстрее. Мы думаем, что, оптимизируя наш процесс разработки вокруг многоразового использования существующего кода, нам не придется писать очень много кода — и мы с меньшей работой и быстрее доведем наш софт до выходной двери, так? Но давайте возьмем простой пример. Группа обслуживания клиентов в Music Corp должна форматировать PDF для обеспечения клиентских счетов-фактур. Еще одна часть системы уже занимается генерацией PDF-файлов: мы производим PDF-файлы на складе для распечатки упаковочных листов, прилагаемых к заказам, поставляемым клиентам, и для отправки поставщикам заявок на размещение заказов.

Следуя цели многоразового использования, наша группа будет ориентирована на существующую возможность генерации PDF. Но эта функциональность в настоящее время выполняется другой группой в другой части организации. Поэтому теперь мы должны координировать работу с ними в части внесения необходимых изменений для поддержки наших функций. Это будет означать, что мы должны просить их делать работу за нас или, возможно, должны будем вносить изменения сами и отправлять запрос на включение модифицированного кода (при условии, что наша компания работает таким образом). В любом случае, мы должны координировать свои действия с другой частью организации при внесения изменения.

Мы будем тратить время на координирование с другими людьми и обеспечивать внесение изменений, и все это ради того, чтобы мы могли внедрить наши изменения. Но мы выяснили, что на самом деле мы могли бы намного скорее написать свою собственную имплементацию и отправить функцию клиенту быстрее, чем потратив время на адаптацию существующего кода. Если ваша фактическая цель — меньшее время выхода на рынок, то это будет правильным выбором. Но если вы выполняете оптимизацию с целью многократного использования, надеясь на то, что вы получите более короткое время выхода на рынок, то в итоге вы придете к тому, что будете делать вещи, которые вашу работу замедляют.

Измерять многократное использование в сложных системах трудно, и, как я уже отмечал, оно, как правило, привлекается для достижения чего-то другого. Вместо этого потратьте свое время, сосредоточившись на реальной цели, и признайте, что многоразовое использование не всегда бывает правильным ответом.

---

[7] Термин "выгорающая платформа" (burning platform), как правило, используется для обозначения технологии, которая, по общему мнению, находится в конце своего срока эксплуатации. Получить поддержку со стороны технологии может оказаться слишком тяжело или дорого, а нанять людей с соответствующим опытом — слишком трудно. Распространенный пример технологии, которая, по общему мнению большинства организаций, считается выгорающей платформой, — мейнфреймовое приложение на COBOL.

Когда микрослужбы будут плохой идеей?

Мы потратили годы на разведывание потенциальных выгод от архитектур на основе микрослужб. Но в некоторых ситуациях я рекомендую вам не использовать микрослужбы вообще. Давайте теперь рассмотрим несколько таких ситуаций.

Неясный домен

Неправильное понимание контуров служб приводит к дорогостоящим результатам. Оно приводит к большему числу изменений между службами, чрезмерно сопряженным компонентам и в целом будет хуже, чем просто иметь единую монолитную систему. В книге "Создание микросервисов" я поделился опытом работы коллектива SnapCI в ThoughtWorks. Несмотря на то что они очень хорошо знали домен (область) непрерывной интеграции, их первоначальная попытка в создании контуров служб для их технического решения по непрерывной интеграции на хосте (hosted-CI) была не совсем правильной. Она привела к высокой стоимости изменений и высокой стоимости владения. После нескольких месяцев борьбы с этой проблемой указанная группа решила снова объединить службы в одно большое приложение. Позже, когда функциональный набор приложения несколько стабилизировался и группа получила более четкое представление о домене, отыскать эти стабильные контуры стало легче.

SnapCI был инструментом непрерывной интеграции и непрерывной доставки, размещаемым на хосте. Ранее данная группа работала над другим подобным инструментом, Go-CD, который теперь служит средством непрерывной доставки с открытым исходным кодом и разворачивается локально, а не в облаке. Хотя между проектами SnapCI и Go-CD было некоторое повторное использование кода, в конце концов, SnapCI оказался совершенно новой кодовой базой. Тем не менее предыдущий опыт группы в домене обеспечения инструментарием для непрерывной доставки (CD) вдохновил их двигаться быстрее в определении контуров и создании своей системы в виде набора микрослужб.

Однако через несколько месяцев стало ясно, что варианты использования SnapCI тонко отличались настолько, что первоначальный подход к контурам служб был не совсем правильным. Он приводил к большому числу изменений, вносимых во всех службах, и ассоциированной высокой стоимости изменений. В конце концов, группа объединила службы обратно в одну монолитную систему, что дало им время получше разобраться в том, где должны проходить контуры. Через год группе удалось разложить монолитную систему на микрослужбы, контуры которых оказались гораздо стабильнее. Это далеко не единственный пример такой ситуации, который я видел. Бывает, что преждевременная декомпозиция системы на микрослужбы становится дорогостоящей, в особенности если вы являетесь новичком в затрагиваемом домене. Во многих отношениях иметь существующую кодовую базу, которую вы хотите разложить на микрослужбы, намного проще, чем пытаться перейти к микрослужбам с самого начала.

Если вы чувствуете, что у вас еще нет полного представления о вашем домене, то правильнее будет решить этот вопрос до того, как приступить к декомпозиции сис-

темы. (Еще одна причина для того, чтобы заняться доменным моделированием! Вскоре мы обсудим его подробнее.)

Стартапы

Эта ситуация покажется немного спорной, т. к. многие организации, известные своим использованием микрослужб, считаются стартапами, но на самом деле многие из этих компаний, включая Netflix, Airbnb и т. п., перешли на архитектуру, основанную на микрослужбах, позже в ходе своей эволюции. Микрослужбы являются отличным вариантом для "масштабирования вверх" — стартапных компаний, которые до этого утвердили, по крайней мере, основы соответствия продукта рынку[8] и теперь занимаются масштабированием с целью увеличения (или, скорее всего, просто достижения) прибыльности.

Стартапы, в отличие от масштабирования вверх, часто экспериментируют с различными идеями в попытке отыскать степень соответствия с клиентами. Это приводит к огромным сдвигам в первоначальном видении продукта, по мере разведывания пространства, в результате давая огромные сдвиги в домене продукта.

Настоящий стартап — это, скорее всего, небольшая организация с ограниченным финансированием, которая должна сосредоточивать все свое внимание на отыскании верной степени соответствия для своего продукта. Микрослужбы в первую очередь решают как раз те виды проблем, которые возникают у стартапов, когда они находят ту самую степень соответствия с их клиентской базой. Другими словами, микрослужбы — отличный способ решения тех видов проблем, которые вы приобретете, как только у вас будет первоначальный успех в качестве стартапа. Поэтому изначально сосредоточьтесь на успехе! Если ваша первоначальная идея плоха, то не имеет значения, создали ли вы ее с помощью микрослужб или без них.

Гораздо проще подразделить существующую "браунфилдовскую"[9] систему, чем сделать это непосредственно с новой "гринфилдовской" системой, которую создал бы стартап. У вас есть еще над чем поработать. У вас есть код, который вы можете проэкзаменовать, и вы можете поговорить с людьми, которые используют и сопровождают систему. Вы также знаете, как выглядит хорошее, — у вас есть рабочая система, готовая к изменению, облегчая вам понимание того, когда вы допустили ошибку или были слишком агрессивны в процессе принятия решений.

У вас также есть система, которая выполняет работу фактически. Вы понимаете, как она работает и как она ведет себя в производстве. Декомпозиция на микрослужбы, например, вызывает некоторые неприятные проблемы с производительно-

---

[8] Соответствие продукта рынку (product/market fit) — это степень, в которой продукт удовлетворяет сильный рыночный спрос. Определяется как первый шаг к созданию успешного бизнеса, на котором компания встречается с ранними последователями, собирает обратную связь и измеряет интерес к своему продукту(ам). См. **https://en.wikipedia.org/wiki/Product/market_fit**, а также **https://medium.com/lean-digital/что-такое-market-fit-или-почему-нет-продаж-27efad0b0d9** — *Пер.*

[9] "Браунфилдовская" система (brownfield system) — это система, имплементируемая на основе существующей рабочей системы и каких-либо ее объектов — *Пер.*

стью, но с "браунфилдовской" системой у вас есть шанс утвердить здоровый базовый уровень, прежде чем вносить изменения, потенциально влияющие на производительность.

Я, конечно, не говорю никогда не заниматься микрослужбами для стартапов; я лишь хочу донести до вас следующее: смысл этих факторов в том, что вы должны быть осторожными. Проводите раздел только вокруг тех контуров, которые ясны изначально, и держите остальные на более монолитной стороне. Это также даст вам время оценить пределы того, насколько вы являетесь зрелыми с операционной точки зрения, — если вы с трудом справляетесь с управлением двумя службами, то управление десятью обязательно будет трудным.

Софт инсталлируется и управляется клиентом

Если вы создаете софт, упаковываемый и поставляемый клиентам, которые затем управляют им сами, то микрослужбы будут плохим вариантом выбора. Когда вы мигрируете на архитектуру, основанную на микрослужбах, вы заталкиваете большой объем сложности в операционный домен. Предыдущие методы, использовавшиеся для мониторинга и устранения неполадок монолитного развертывания, могут и не работать с новой распределенной системой. В свою очередь, группы, которые предпринимают миграцию на микрослужбы, компенсируют эти трудности, осваивая новые навыки или, возможно, внедряя новую технологию, и это не то, что вы в типичной ситуации ожидаете от своих конечных клиентов.

В типичной ситуации с софтом, устанавливаемым клиентом, вы ориентируетесь на определенную платформу. Например, вы можете сказать "требуется сервер Windows 2016" или "требуется macOS 10.12 или выше". Это четко определенные целевые среды развертывания, и вы, вполне возможно, упаковываете свой монолитный софт с использованием механизмов, хорошо понятных людям, которые управляют этими системами (например, доставка служб Windows, укомплектованных в пакет установщика Windows Installer). Ваши клиенты, наверняка, знакомы с покупкой и запуском софта таким образом.

Представьте себе, какие у вас будут проблемы, если вместо передачи им одного выполняемого и управляемого ими процесса вы станете передавать им 10 или 20? Или, возможно, еще агрессивнее, ожидая, что они будут выполнять ваш софт на кластере Kubernetes или на чем-то аналогичном?

Реальность такова, что нельзя ожидать, что у ваших клиентов будут иметься навыки или платформы для управления архитектурой на основе микрослужб. Даже если они их будут иметь, то у них не будет тех навыков или платформ, которые вам нужны. Например, существует большая разница между версиями инсталляций Kubernetes.

Отсутствует веская причина!

И, наконец, категорически не следует внедрять микрослужбы, если у вас нет четкого представления о том, что именно вы пытаетесь достичь. Как мы увидим, результат, который вы ищете от внедрения вами микрослужб, будет определять, где вы

начинаете эту миграцию и как вы осуществляете декомпозицию системы. Не имея четкого представления о своих целях, вы блуждаете в темноте. Внедрять микрослужбы только потому, что все остальные это делают, — идея просто ужасная.

Компромиссы

До сих пор я описывал причины, по которым люди внедряют микрослужбы изолированно, и изложил (хотя и кратко) аргументы в пользу рассмотрения других вариантов. Однако в реальном мире люди часто пытаются изменить не одну вещь, а сразу много. Это приводит к путанице приоритетов, быстро увеличивая объем необходимых изменений и задерживая получение каких-либо выгод.

Все начинается достаточно невинно. Нам нужно выполнить перепланировку архитектуры нашего приложения, чтобы справляться с существенным ростом трафика, и мы решаем, что микрослужбы — это путь вперед. Кто-то другой подходит и говорит: "Ну, если мы беремся за микрослужбы, то одновременно с этим мы могли бы сделать наши группы автономнее!". Еще один вмешивается и говорит: "И это дает нам отличный шанс попробовать Kotlin в качестве языка программирования!". Прежде чем вы придете в себя, у вас уже будет масштабная инициатива по изменению, которая пытается внедрить автономию групп, масштабировать приложение и одновременно внедрить новую технологию, а также другие вещи, которые люди накинули к программе работы "для верности".

Более того, в этой ситуации микрослужбы становятся запертыми как подход. Если вы сосредоточитесь только на аспекте масштабирования, то во время миграции вы, возможно, осознаете, что вам будет лучше просто промасштабировать существующее монолитное приложение горизонтально. Но это не поможет новым вторичным целям улучшения автономии групп или привлечения Kotlin в качестве языка программирования.

Следовательно, важно отделить мотивирующий стержневой мотив от любых второстепенных выгод, которые вы также хотели бы получить. В этом случае умение обращаться с улучшенным масштабом приложения — самый важный мотив. Работы, проделываемые для достижения прогресса по другим второстепенным целям (например, улучшение автономии групп), будут полезными, но если они мешают или отвлекают от ключевого целевого критерия, то они должны отойти на задний план.

Здесь важно осознать, что некоторые вещи важнее других. В противном случае вы не сможете надлежаще расставить приоритеты. Одно из усилий, которое мне здесь нравится, — это думать о каждом вашем желаемом результате, как о ползунковом регуляторе. Каждый ползунок начинается в середине. Когда вы придаете одной вещи б*о*льшую важность, должны опустить приоритет у другой — данный пример можно увидеть на рис. 2.1. Этим четко артикулируется то, например, что, хотя вы хотели бы упростить полиглотное программирование, оно не столь же важно, как обеспечение отказоустойчивости приложения. Что касается выяснения способа ва-

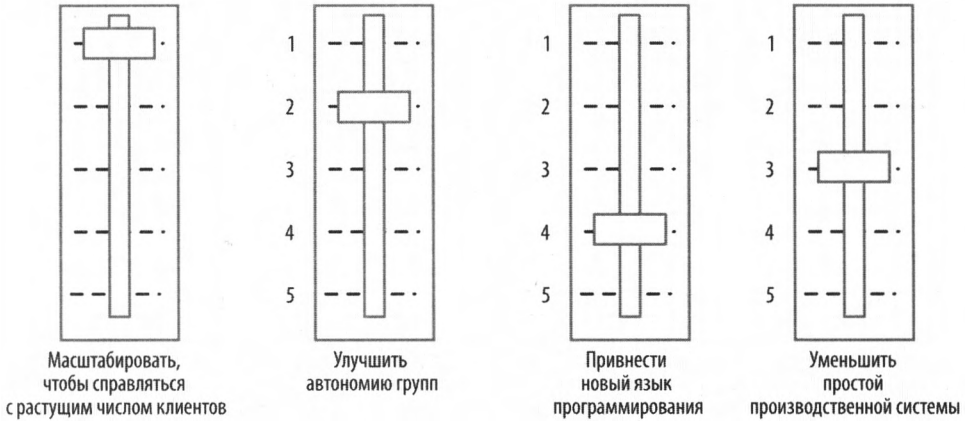


Рис. 2.1. Использование ползунков для балансировки конкурирующих приоритетов, которые у вас могут быть

шего продвижения вперед, то наличие этих четко сформулированных и ранжированных результатов намного упростит принятие решений.

Эти относительные приоритеты могут меняться (и должны меняться по мере того, как вы узнаете больше). Но они помогут и в принятии решений. Если вы хотите распределить ответственность, передавая больше полномочий новым автономным группам, то простые модели, подобные этой, помогут информировать их о локальном принятии решений и сделать более качественный выбор, выровненный по тому, что вы пытаетесь достичь в компании.

Приглашение людей в "путешествие"

Меня часто спрашивают: "Как продать микрослужбы своему боссу?". Этот вопрос, как правило, исходит от разработчика — того, кто увидел потенциал архитектуры, основанной на микрослужбах, и убежден, что этот путь ведет вперед.

Нередко, когда люди расходятся во мнениях по поводу того или иного подхода, это происходит потому, что у них разные взгляды на то, чего вы пытаетесь достичь. Важно, чтобы вы и другие люди, которых вам нужно взять с собой в "путешествие", имели общее понимание того, чего вы пытаетесь достичь. Если ваши взгляды в этом совпадают, то, по крайней мере, вы знаете, что вы не согласны только в том, как туда добраться. Поэтому все снова возвращается к цели — если другие люди в организации разделяют общую цель, то они с гораздо большей вероятностью будут готовы к внесению изменений.

На самом деле стоит подробнее разведать вопрос о том, каким образом можно и продать идею, и воплотить ее в жизнь, черпая вдохновение в одной из наиболее известных моделей, помогающих вносить организационные изменения. Давайте взглянем на нее.

Изменение организаций

Восьмиступенчатый процесс д-ра Джона Коттера (John Kotter) по имплементированию изменений в организации лежит в основе работы менеджеров по изменениям по всему миру, отчасти потому, что он хорошо справляется с перегонкой требуемых действий в дискретные, понятные шаги. Эта модель далеко не единственная, но она является именно той, которую я нахожу наиболее полезной.

Этот процесс, очерченный на рис. 2.2, описывается огромным объемом информационного материала, поэтому я не буду здесь останавливаться на указанном процессе слишком много[10]. Тем не менее, стоит кратко описать шаги и подумать о том, как они нам помогут, если мы подумываем о принятии на вооружение архитектуры, основанной на микрослужбах.

Рис. 2.2. Восьмиступенчатый процесс Коттера для осуществления организационных изменений

Прежде чем я здесь обрисую указанный процесс, должен отметить, что эта модель изменений обычно используется для учреждения крупномасштабных организационных сдвигов в поведении. Как таковая, она вполне может оказаться огромным перебором, если вы пытаетесь привнести микрослужбы в группу всего из 10 человек. Однако даже в этих условиях с меньшим охватом я нашел эту модель полезной, в особенности на более ранних шагах.

Закрепление чувства насущной необходимости

Возможно, люди посчитают вашу идею перехода на микрослужбы хорошей. Проблема в том, что ваша идея — всего лишь одна из многих хороших идей, которые,

[10] Модель изменений Джона Коттера подробно изложена в его книге "Во главе перемен" (Leading Change, Harvard Business Review Press, 1996).

вероятно, витают в окружении организации. Хитрость заключается в том, чтобы помочь людям понять, что сейчас наступило самое время осуществить это конкретное изменение.

Здесь поможет поиск моментов "открытости к обучению". Иногда самое подходящее время, для того чтобы запереть дверь конюшни, наступает после того, как лошадь была заперта, потому что люди вдруг начинают осознавать, что именно о бегстве лошадей должны быть все их мысли, и теперь они начинают осознавать, что у них даже есть дверь, и "Вон, смотрите, ее можно закрыть и все такое!". В моменты после преодоления кризиса у вас есть краткий миг в сознании людей, где срабатывает толчок к изменениям. Прождете слишком долго, и боль — и причины этой боли — спадут.

Помните, что своими действиями вы не пытаетесь сказать: "Нам прямо сейчас следует заняться микрослужбами!". Вы стараетесь поделиться чувством насущной необходимости в том, чего вы хотите достичь, — и, как я уже сказал, микрослужбы не являются целью!

Создание направляющей коалиции

Вам не обязательно интегрировать всех, но их должно быть достаточно, для того чтобы это произошло. Вам нужно выявить людей из вашей организации, которые помогут вам продвигать это изменение вперед. Если вы являетесь разработчиком, то первые из них, вероятно, будут среди ваших ближайших коллег в вашей группе и, возможно, кто-то более старший по должности — это будет технический руководитель, архитектор или менеджер по доставке. В зависимости от масштаба изменения участие большого числа людей может не понадобиться. Если вы меняете просто то, как ваша группа что-то делает, то соответствующего "прикрытия с воздуха" может хватить. Если вы пытаетесь изменить способ разработки софта в вашей компании, то для обеспечения поддержки вам понадобится кто-то на уровне сотрудников исполнительного звена (возможно, директор по информационным технологиям (CIO) или технический директор (CTO).

Интегрирование людей с целью помочь осуществить это изменение не будет проходить гладко. Независимо от того, насколько хороша эта идея, если человек никогда о вас не слышал или никогда с вами не работал, то почему он должен поддерживать вашу идею? Доверие завоевывается. Кто-то с гораздо большей вероятностью поддержит вашу большую идею, если он уже работал с вами и достиг меньших, быстрых побед.

Здесь важно, чтобы у вас были вовлечены люди, не связанные с доставкой софта. Если вы уже работаете в организации, где барьеры между "ИТ" и "Бизнесом" были разрушены, то все, вероятно, будет ОК. С другой стороны, если обособленность подразделений все еще существует, то вам, возможно, придется протянуть руку через проход, для того чтобы найти сторонника в другом месте. Конечно, если ваша архитектура на основе микрослужб ориентирована на решение проблем, с которыми организация сталкивается в своей деятельности, то продать ее будет намного проще.

Причина, по которой вам потребуется участие людей за пределами обособленного ИТ-подразделения, заключается в том, что многие изменения, которые вы осуществляете, потенциально могут значительно повлиять на работу и поведение софта. Вам нужно будет принять разные компромиссы о том, например, как ваша система ведет себя во время режимов сбоя или как вы справляетесь с задержкой. Например, кэширование данных во избежание вызова службы, — хороший способ уменьшить задержку ключевых операций в распределенной системе, но компромисс как раз в том и состоит, что в результате этого пользователи вашей системы увидят несвежие данные. Разве это правильно? Вероятно, вам придется обсудить этот вопрос с вашими пользователями, и эта дискуссия будет трудной, если люди, которые в вашей организации отстаивают интересы ваших пользователей, не понимают причины изменения.

Развитие видения и стратегии

Это важно там, где вы собираете своих людей вместе и договариваетесь о том, какие изменения вы надеетесь принести (видение) и как вы собираетесь туда попасть (стратегия). Видение изменений — скользкая штука. Оно должно быть реалистичным, но в то же время желательным, и ключом является отыскание баланса между этим. Чем шире видение распространено, тем больше работы пойдет на его упаковку для интегрирования людей. Но видение может быть расплывчатым и все равно работать с меньшими группами ("нам нужно уменьшить количество багов!").

Видение в основном касается цели — того, к чему вы стремитесь. Стратегия имеет дело с тем, как ее добиться. Микрослужбы собираются достичь этой цели (вы надеетесь; они будут частью вашей стратегии). Помните, что ваша стратегия может поменяться. Важно оставаться приверженным своему видению, но чрезмерная приверженность определенной стратегии перед лицом фактов, подтверждающих обратное, таит опасность и приводит к значительным понесенным расходам.

Коммуницирование видения перемен

Вероятно, здорово иметь большое видение перемен, но не делайте его настолько большим, что люди не поверят в их осуществление. Недавно я увидел заявление генерального директора (CEO) крупной организации, в котором говорилось (несколько перефразируя) следующее:

> "В ближайшие 12 месяцев мы снизим затраты и будем осуществлять доставки быстрее, благодаря переходу на микрослужбы и внедрению облачных технологий".
>
> *– Неназванный генеральный директор*

Никто из сотрудников компании, с которыми я разговаривал, не верил, что подобное возможно. В приведенном заявлении часть проблемы кроется в том, что в нем заявлены потенциально противоречивые цели, которые изложены в общих чертах: полномасштабное изменение способа доставки софта поможет вам поставлять его быстрее, но осуществление этого плана за 12 месяцев не приведет к снижению за-

трат, поскольку вам придется привлечь новые навыки, и вы будете испытывать негативное влияние на продуктивность до тех пор, пока новые навыки не будут заложены в основу. Другой вопрос — это упомянутые здесь временны́е рамки. В данной конкретной организации скорость изменений была достаточно малой, чтобы считать 12-месячную цель смехотворной. Поэтому любое видение, которое вы разделяете, должно быть в какой-то степени правдоподобным.

Что касается распространения своего видения, то следует начинать с малого. Много лет назад я работал в рамках программы под названием "Легионеры тестирования" (Test Mercenaries), чтобы оказать помощь во внедрении практики автоматизации тестирования в Google. Причиной запуска этой программы явились предыдущие попытки со стороны тех, кого мы сейчас называем сообществом практиков ("групплетов" в номенклатуре Google), с целью помочь распространить понимание важности автоматизированного тестирования. Одной из первых попыток программы по распространению информации о тестировании была инициатива под названием "Тестирование на унитазе". Она состояла из коротких одностраничных статей, прикалываемых к дверям туалета, с тем чтобы люди могли читать их "на досуге"! Я не предлагаю, что эта технология будет работать везде, но она хорошо работала в Google — и это был действительно эффективный способ распространения небольших действенных советов[11].

И последнее замечание. Наблюдается тенденция все большего ухода от общения "лицом к лицу" в пользу таких систем, как Slack. В том что касается обмена важными сообщениями такого рода, то общение "лицом к лицу" (в идеале лично, но, возможно, через видеосвязь) будет значительно эффективнее. Оно облегчает понимание реакции людей на эти идеи, а также помогает калибровать ваше сообщение и избегать недоразумений. Даже если для транслирования своего видения через более крупную организацию вам понадобятся другие формы коммуникации, сначала пообщайтесь как можно больше лицом к лицу. Это поможет вам гораздо эффективнее улучшить свой "месседж".

Расширение полномочий сотрудников по широкому кругу действий

"Расширение полномочий сотрудников" — это призыв со стороны консалтинга по менеджменту поспособствовать им в выполнении своей работы. Чаще всего он означает нечто довольно простое — удаление "дорожных пробок".

Вы поделились своим видением и подняли воодушевление — и что затем происходит? На пути начинают вставать всяческие помехи. Одна из наиболее распространенных проблем кроется в том, что люди слишком заняты тем, что они делают сейчас, а тут еще надо иметь пропускную способность для осуществления перемен —

---

[11] Более подробная история инициативы по внесению изменений с целью внедрения тестирования в Google изложена в отличной тематической подборке Майка Блэнда (**http://bit.ly/2omkxVy**), которую стоит прочитать. Майк также подробно описал историю "тестирования на унитазе" (**http://bit.ly/2ojpWwm**).

именно поэтому компании часто привлекают в организацию новых людей (возможно, через наем или в качестве консультантов) с целью дать группам дополнительную пропускную способность и опыт в осуществлении изменений.

В качестве конкретного примера в том, что касается внедрения микрослужб, бывает, что существующие процессы по обеспечению инфраструктурой становятся реальной проблемой. Если способ, с помощью которого ваша организация проводит развертывание новой производственной службы, предусматривает размещение заказа на вычислительное оборудование за шесть месяцев вперед, то внедрение технологии, позволяющей предоставлять виртуализированные среды выполнения по требованию (например, виртуальные машины или контейнеры), станет огромным благом, как и сдвиг в сторону поставщика публичного облака.

Однако я хочу повторить свой совет из предыдущей главы. Ради этого не стоит бросать в смесь новую технологию. Привлекайте ее для того, чтобы решать конкретные проблемы, которые вы видите. По мере выявления препятствий внедряйте новые технологии и устраняйте эти проблемы с их помощью. Не попадайте в ловушку, тратя год на определение *Идеальной платформы для микрослужб* только для того, чтобы в итоге обнаружить, что она на самом деле не решает проблемы, которые у вас есть.

В рамках программы "Легионеры тестирования" в Google мы изготовили каркасы, облегчающие создание и управление тестовыми наборами, повысили степень видимости тестов в рамках системы ревизии кода и даже в итоге поспособствовали созданию нового общекорпоративного инструмента CI, упрощающего выполнение тестов. Но мы делали все это не сразу. Мы работали с несколькими группами, видели болевые точки, учились на них, а затем вкладывали время в привлечение новых инструментов. Мы также начали с малого — процесс изготовления тестового набора был довольно простым, но работа по изменению системы ревизии кода в масштабах всей компании была гораздо крупнее. Мы не пытались решать его до тех пор, пока не добились некоторых успехов в других местах.

Генерирование краткосрочных выигрышей

Если людям потребуется слишком много времени на то, чтобы увидеть прогресс, то они потеряют веру в видение перемен. Поэтому идите на несколько быстрых побед. Концентрация с самого начала на малых, легких, "низко висящих фруктах" поможет создать импульс. В том что касается декомпозиции на микрослужбы, то функциональность, которую можно легко извлечь из нашего монолита, должна находиться в вашем списке высоко. Но, как мы уже установили, сами по себе микрослужбы не являются целью, поэтому вам нужно будет уравновешивать легкость извлечения некоторой части функциональности относительно выгоды, которую она принесет. Мы вернемся к этой идее позже в данной главе.

Разумеется, если вы выберете нечто, что считаете легким, и в итоге столкнетесь с огромными трудностями, то это станет источником ценного понимания вашей стратегии и заставит вас пересмотреть то, что вы делаете. Это совершенно нор-

мально! Главное сначала сосредоточиться на чем-то легком, и тогда, скорее всего, понимание будет получено раньше. Нам свойственно ошибаться, мы способны лишь структурировать вещи, благодаря чему обеспечиваем свое максимально скорое обучение на этих ошибках.

Консолидация выигрышей и порождение новых изменений

Как только у вас есть некий успех, важно не почивать на лаврах. Быстрые выигрыши останутся единственными, если вы не будете продолжать настаивать на своем. После успехов (и неудач) важно делать паузы и обдумывать их в плане того, как продолжать осуществлять изменения. Возможно, когда вы достигнете разных частей организации, вам придется поменять свой подход.

Во время транзита на микрослужбы по мере того как вы будете вгрызаться все глубже и глубже, вы обнаружите, что продвигаться становиться все труднее. Возможно, вопрос улаживания декомпозиции базы данных вы откладывали с самого начала, но его не получится откладывать вечно. Как мы разведаем в *главе 4*, в вашем распоряжении имеется целый ряд методов, но, для того чтобы выяснить, какой подход является правильным, потребуется тщательный анализ. Просто помните, что метод декомпозиции, который работал для вас в одной области вашего монолита, может не работать где-то еще — вам нужно будет постоянно пробовать новые способы продвижения вперед.

Заякоревание новых подходов в культуре

Продолжая выполнять итерации, внедрять изменения и делиться историями успехов (и неудач), новый способ работы начнет становиться обычным делом. Крупная его часть касается обмена историями с вашими коллегами, с другими группами и другими людьми в организации. Слишком часто, решив трудную задачу, мы просто двигаемся дальше к следующей. Для того чтобы перемены масштабировались и закреплялись — необходимо постоянно отыскивать способы делиться информацией внутри организации.

Со временем новый способ делать что-то становится общепринятым. Если вы посмотрите на компании, которые далеко продвинулись по пути внедрения архитектур, основанных на микрослужбах, то вопрос о том, был ли этот подход правильным, давно снят с повестки дня. Сейчас дела делаются именно так, и организация понимает, как их делать хорошо.

Это, в свою очередь, может создать новую проблему. Как только большая новая идея становится установившимся способом работы, то как быть уверенным в том, что последующие, более совершенные подходы будут иметь пространство для своего появления и, может, даже вытеснят то, как все делается сейчас?

Важность поступательной миграции

> "Если вы переписываете методом “Большого взрыва”, то единственное, что вам гарантировано, — это большой взрыв".
>
> *– Мартин Фаулер (Martin Fowler)*

Если вы уже готовы решить, что декомпозиция существующей монолитной системы является правильным поступком, то я настоятельно рекомендую вам "откалывать" от этих монолитов, вынимая каждый раз по "кусочку". Поступательный подход поможет вам усваивать знания о микрослужбах по мере продвижения работы, а также ограничит влияние ошибок (и, поверьте, они у вас будут!). Думайте о монолите как о "мраморной глыбе". Можно было бы просто взорвать всю глыбу целиком, но это редко заканчивается хорошо. Гораздо больше смысла в откалывании от нее постепенно, поступательно, по кусочку.

Трудность в том, что стоимость эксперимента по переходу с нетривиальной монолитной системы на архитектуру, основанную на микрослужбах, будет большой, и если вы делаете все сразу, то бывает непросто получить хорошую обратную связь о том, что работает (или не работает) хорошо. Гораздо легче разбить такое "путешествие" на более мелкие этапы; каждый из них можно проанализировать и из каждого извлечь уроки. Именно по этой причине я был большим поклонником итеративной доставки софта еще до появления концепции маневренности agile, т. е. принятия того факта, что я буду делать ошибки и поэтому мне нужен способ уменьшить размер этих ошибок.

Любой транзит на архитектуру, основанную на микрослужбах, должен учитывать эти принципы. Разбейте большое путешествие на множество малых шагов. Каждый шаг можно выполнить и усвоить. Если этот шаг окажется ретроградным, то совсем небольшим. В любом случае вы на нем научитесь, и следующий шаг, который вы сделаете, будет проинформирован теми шагами, которые были сделаны раньше.

Как мы уже говорили ранее, разбиение вещей на более мелкие кусочки также позволяет выявлять быстрые выигрыши и на них обучаться. Это поможет упростить следующий шаг и накопить импульс. Выделяя микрослужбы по одной, вы также выводите наружу запертую в них ценность, приносимую ими поступательно, вместо ожидания какого-то развертывания методом "Большого взрыва".

Все это приводит к тому, что стало почти заезженным советом для людей, которые поглядывают на микрослужбы. Если вы считаете, что эта идея хорошая, то начните с малого. Выберите одну или две области функциональности, имплементируйте их в качестве микрослужб, разверните их в производство и отрефлексируйте их успешность. Далее в этой главе я покажу вам модель, помогающую выяснить, с каких микрослужб следует начинать.

Главное — производство

Действительно, важно отметить, что извлечение микрослужбы не может считаться завершенным до тех пор, пока та не находится в производстве и активно не используется. Одна из целей поступательного извлечения состоит в том, чтобы дать нам

возможность учиться на самой декомпозиции и понимать ее влияние. Подавляющее большинство важных уроков не будет усвоено до тех пор, пока ваша служба не попадет в производство.

Бывает, что декомпозиция на микрослужбы вызывает проблемы с устранением неполадок, трассировкой потоков, задержкой, ссылочной целостностью, каскадными сбоями и целым рядом других вещей. Большинство этих проблем вы заметите только после того, как дойдете до производства. В следующих главах мы рассмотрим методы, которые позволяют осуществлять развертывание в производственной среде, но ограничивают влияние проблем, когда те возникают. Если вы вносите малое изменение, то гораздо легче засекать (и исправлять) проблему, которую вы создаете.

Стоимость изменения

Существует много причин, по которым на протяжении всей книги я пропагандирую необходимость внесения малых, поступательных изменений, но один из ключевых мотивов — понимание влияния каждого осуществляемого нами изменения и смена курса, если это необходимо. Это позволяет нам лучше оценивать стоимость ошибки, но не устраняет вероятность ошибок полностью. Мы можем и будем совершать ошибки, и мы должны это осознавать. Однако мы также должны понимать, как лучше всего смягчать издержки, связанные с этими ошибками.

Обратимые и необратимые решения

Джефф Безос (Jeff Bezos), генеральный директор Amazon, в своих ежегодных письмах акционерам дает интересное представление о том, как работает Amazon. Письмо за 2015 год содержало следующий пассаж:

> "Некоторые решения являются последовательными и необратимыми или почти необратимыми — однопутные двери — и эти решения должны приниматься методично, тщательно, медленно, с большим обдумыванием и консультацией. Если вы проходите через них и вам не нравится то, что вы видите на другой стороне, то вы не сможете вернуться туда, где вы были раньше. Назовем эти решения решениями 1-го рода. Но большинство решений не таковы — они изменчивы, обратимы — они являются "двупутными дверьми". Если вы приняли субоптимальное решение 2-го рода, то вам не придется жить с последствиями так долго. Вы можете снова открыть дверь и пройти обратно. Решения 2-го рода могут и должны приниматься быстро людьми с высоким уровнем суждений или малыми группами".
>
> – *Джефф Безос*, Письмо акционерам Amazon (2015)

Безос продолжает: "Люди, которые не принимают решения, часто попадают в ловушку, рассматривая решения 2-го рода как решения 1-го рода. Все становится вопросом жизни или смерти, все становится важнейшим мероприятием. Проблема в том, что внедрение архитектуры, основанной на микрослужбах, приносит с собой массу вариантов относительно того, как вы делаете вещи, а это означает, что вам

потребуется принимать гораздо больше решений, чем раньше. И если вы (или ваша организация) к этому не привыкли, то вы окажетесь в этой ловушке, и прогресс застопорится".

Указанные термины не особо описательны, и бывает трудно вспомнить, что на самом деле означает решение 1-го или 2-го рода, поэтому я предпочитаю имена "необратимое" (для решения 1-го рода) или "обратимое" (для решения 2-го рода)[12].

Хотя эта концепция мне нравится, я не думаю, что решения всегда аккуратно попадают в одну из этих двух корзин; данный вопрос видится немного тоньше, чем тут. Я бы предпочел думать в терминах необратимости и обратимости как находящихся на двух концах спектра, как показано на рис. 2.3.

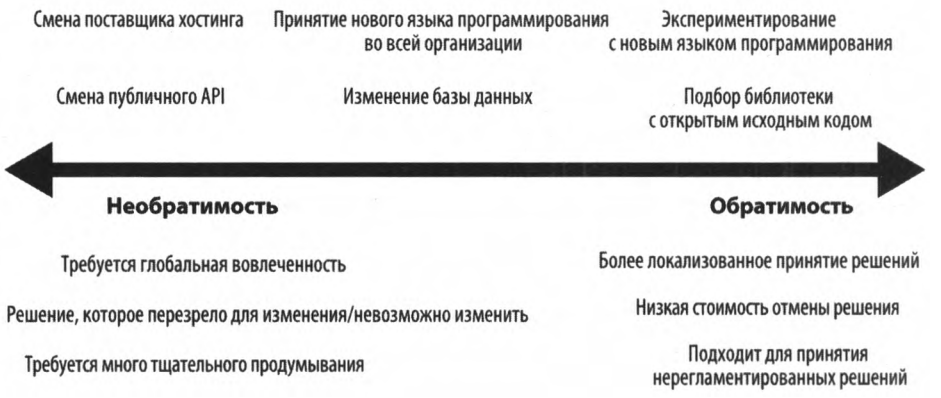


Рис. 2.3. Спектр различий между необратимыми и обратимыми решениями с примерами

Оценить то, где вы находитесь в этом спектре, с самого начала будет трудно, но в сущности все это возвращается к пониманию влияния, в случае если позже вы решите изменить свое мнение. Чем большее влияние окажет последующее исправление курса, тем больше это решение начинает выглядеть как необратимое.

Реальность такова, что огромное число решений, которые вы примете в рамках транзита на микрослужбы, будет направлено в сторону обратимого конца спектра. Софт имеет характерное свойство, где часто возможны откаты или отмены, вы можете откатить изменение или развертывание софта. Вам в первую очередь следует учитывать именно стоимость изменения вашего мнения в дальнейшем.

Необратимые решения потребуют большей вовлеченности, тщательного обдумывания и учета всех "за" и "против", и вам следует (должным образом) уделять этому больше времени. Чем дальше мы продвигаемся по этому спектру к нашим обратимым решениям, тем больше мы можем просто положиться на наших коллег, которые близки к затрагиваемой проблеме, и принять верное решение, зная, что если они принимают неправильное решение, то позже его будет легко исправить.

[12] Чаевые в шляпу Мартину Фаулеру в благодарность за имена!

Более легкие места для эксперимента

Стоимость, связанная с переносом кода в пределах кодовой базы, довольно мала. Нас поддерживает большое число инструментов, и если мы создаем проблему, то исправление наступает в общем-то быстро. Однако разложение базы данных требует гораздо большей работы, и откат изменения базы данных так же сложен. Схожим образом распутывание чрезмерно сопряженной интеграции между службами или необходимость полностью переписывать API, используемый многочисленными потребителями, становится значительным мероприятием. Высокая стоимость изменений означает, что эти операции становятся все более рискованными. Как управлять этим риском? Мой подход — пытаться совершать ошибки там, где влияние будет наименьшим.

Я склонен посвящать бо́льшую часть своих мыслей тому месту, где стоимость изменения и ошибок настолько низкие, насколько это возможно: на доске. Набросайте предлагаемый вами дизайн. Посмотрите, что произойдет при прогоне вариантов использования через то, что, по вашему мнению, будет контуром службы. В случае нашего музыкального магазина, например, представьте, что произойдет, когда клиент ищет запись, регистрируется на веб-сайте или покупает альбом. Какие вызовы делаются? Появляются ли у вас странные циклические ссылки? Не замечаете ли вы две чрезмерно "болтливые" службы, что может сигнализировать о том, что они должны быть единым целым?

Так с чего же мы начнем?

ОК, значит, мы поговорили о важности четкого артикулирования наших целей и понимания потенциальных компромиссов, которые могут существовать. Что же дальше? Скажем так, нам нужно представление о том, какие части функциональности мы хотим извлечь в службы. Оно даст нам возможность начать думать рационально о том, какие микрослужбы мы будем создавать дальше. В том, что касается декомпозиции существующей монолитной системы, то для работы нам потребуется некоторая форма логической декомпозиции, и именно здесь пригодится доменно-обусловленный дизайн.

Доменно-обусловленный дизайн

В *главе 1* я представил доменно-обусловленный дизайн, как важную концепцию, помогающую задавать контуры наших служб. Разработка доменной модели также помогает нам в том, что касается выработки способа приоритизации нашей декомпозиции. На рис. 2.4 приведен пример высокоуровневой доменной модели компании Music Corp. Вы видите ряд ограниченных контекстов, выявленных в результате доменного моделирования. Мы ясно видим связи между этими ограниченными контекстами, которые, как мы предполагаем, будут представлять взаимодействия внутри самой организации.

Каждый из этих ограниченных контекстов представляет потенциальную единицу декомпозиции. Как мы обсуждали ранее, ограниченные контексты — отличная от-

правная точка для определения контуров микрослужб. Итак, у нас уже есть список приоритизируемых вещей. Но у нас есть и полезная информация в виде связей между этими ограниченными контекстами, которая помогает нам оценить относительную трудность извлечения разных частей функциональности. Мы вскоре вернемся к этой идее.

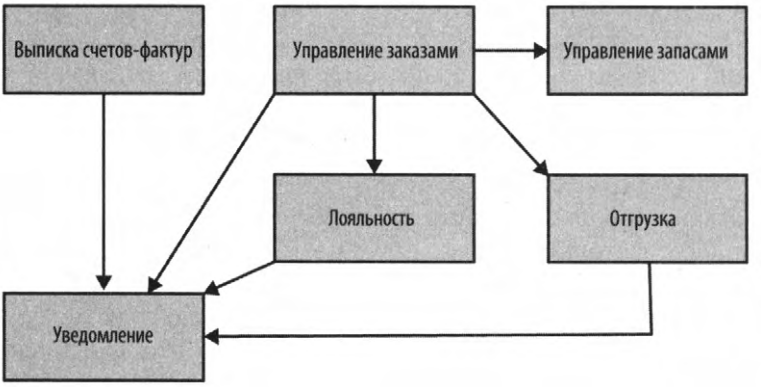


Рис. 2.4. Ограниченные контексты и связи между ними в Music Corp

Я рассматриваю создание доменной модели как почти необходимый шаг, который является частью структурирования транзита на микрослужбы. Нередко вызывает опасение то, что многие люди не имеют прямого опыта в создании такого представления. Они также очень обеспокоены объемом предстоящей работы. Реальность такова, что, хотя в создании логической модели подобного рода наличие опыта значительно помогает, даже небольшой объем усилий приносит некоторые по-настоящему полезные выгоды.

Как далеко заходить?

По мере приближения к декомпозиции существующей системы, ее перспектива выглядит пугающей. Многие люди, вероятно, строили и продолжают строить эту систему, и, по всей вероятности, гораздо более крупная группа людей фактически использует ее изо дня в день. Учитывая масштаб, попытка придумать подробную доменную модель для всей системы выглядит устрашающей.

Важно понимать, что от доменной модели нам нужна лишь информация, представленная в достаточном объеме для принятия разумного решения о том, с чего начинать декомпозицию. Вероятно, у вас уже есть некоторые идеи в отношении тех частей вашей системы, которые нуждаются во внимании больше всего, и, следовательно, для монолита достаточно придумать обобщенную модель с точки зрения высокоуровневых группировок функциональности и выбрать те части, в которые вы хотите погрузиться глубже. Всегда есть опасность, что если смотреть только на часть системы, то можно пропустить более крупные системные вопросы, которые требуют решения. Но я бы не стал на этом зацикливаться — вовсе не требуется делать это правильно с первого раза, вам просто нужна информация в достаточном

объеме для принятия нескольких последующих информированных шагов. Вы можете — и должны — постоянно уточнять доменную модель, по мере того как узнаете больше, и держать ее свежей, отражая новую функциональность по мере ее внедрения.

Событийный штурм

Событийный штурм (event storming), метод, созданный Альберто Брандолини (Alberto Brandolini), представляет собой коллаборативное усилие с участием технических и нетехнических заинтересованных сторон, которые вместе определяют совместную доменную модель. Событийный штурм работает снизу вверх. Участники начинают с определения "доменных событий" — вещей, которые происходят в системе. Затем эти события группируются в агрегаты, а далее агрегаты группируются в ограниченные контексты.

Важно отметить, что событийный штурм не означает, что вы должны затем строить событийно-обусловленную систему. Вместо этого он фокусируется на понимании того, какие (логические) события происходят в системе, — выявлении фактов, которые вас интересуют как участника системы. Эти доменные события отображаются в события, активируемые в рамках событийно-обусловленной системы, но они представляются разными способами.

Одна из вещей, на которых Альберто действительно концентрирует внимание с помощью этого метода, — это идея коллективного определения модели. Результатом данного усилия является не только сама модель, но и совместное понимание модели. Для того чтобы этот процесс работал как надо, необходимо собрать в вашей комнате правильные заинтересованные стороны — и часто в этом состоит самая большая трудность.

Более подробное разведывание темы событийного штурма выходит за рамки этой книги, но указанный метод я использовал, и он мне очень нравится. Если вы хотите узнать больше, то прочитайте статью Альберто "Введение в событийный штурм" (в настоящее время статья пополняется)[13].

Использование доменной модели для приоритизации

Мы можем получить некоторые полезные сведения из диаграмм, таких, как на рис. 2.4. Основываясь на числе вышестоящих или нижестоящих зависимостей, мы можем экстраполировать представление о том, какую функциональность, скорее всего, будет легче или труднее извлечь. Например, если взять извлечение функциональности "Уведомление", то можно ясно увидеть ряд вышестоящих зависимостей, как показано на рис. 2.5 — многие части нашей системы требуют использования этого поведения. Если мы хотим извлечь нашу новую службу "Уведомление", то нам придется проделать большую работу с существующим кодом, изменив вызовы

[13] См. **https://leanpub.com/introducing_eventstorming**.

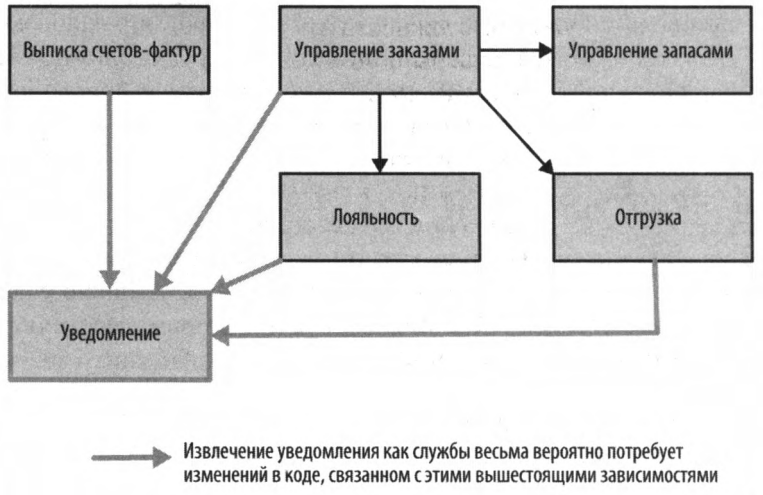


Рис. 2.5. С точки зрения нашей доменной модели функциональность "Уведомления" выглядит логически сопряженной, поэтому извлечь ее будет сложнее

с локальных вызовов существующей уведомительной функциональности и вместо этого сделав их вызовами службы. В *главе 3* мы рассмотрим несколько методов, касающихся такого рода изменений.

Поэтому "Уведомление" не будет хорошим местом для старта. С другой стороны, как показано на рис. 2.6, "Выписка счетов-фактур" вполне может представлять гораздо более простую часть поведения системы, пригодную для извлечения; она не имеет входящих зависимостей, уменьшая необходимые изменения, которые нам нужно внести в существующий монолит. В этих случаях эффективным будет шаблон типа "Фикус-удавка", поскольку мы можем легко перенаправлять входящие вызовы, прежде чем они дойдут до монолита. Мы рассмотрим этот и многие другие шаблоны в следующей главе.

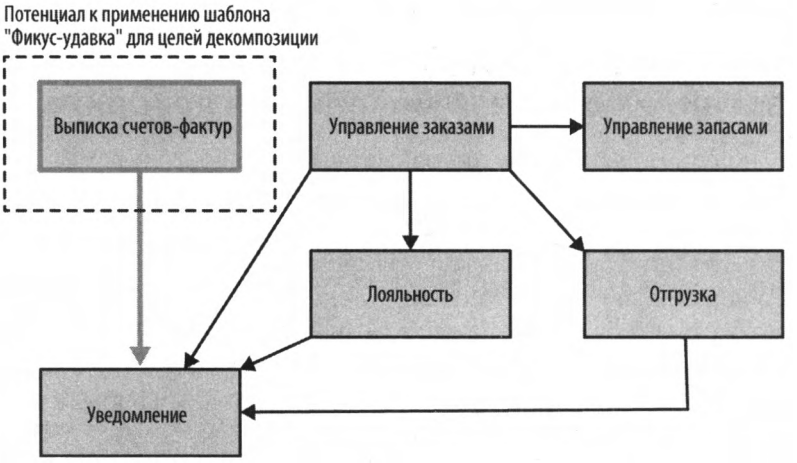


Рис. 2.6. "Выписка счетов-фактур", по всей видимости, извлекается легче

Во время оценивания вероятной сложности извлечения эти связи являются хорошим местом для старта, но мы должны понимать, что доменная модель представляет собой логическую точку зрения на существующую систему. Нет никакой гарантии, что лежащая в основании кодовая структура нашего монолита структурирована именно таким образом. Это означает, что наша логическая модель поможет направлять нас с точки зрения функций, которые, вероятно, будут более (или менее) сопряженными, но для получения более точной оценки степени запутанности текущей функциональности нам все равно потребуется взглянуть на сам код. Доменная модель такого рода не покажет нам, какие именно ограниченные контексты хранят данные в базе данных. Мы, возможно, обнаружим, что "Выписка счетов-фактур" управляет большим объемом данных, и, следовательно, нам придется учитывать влияние работы по декомпозиции базы данных. Как мы обсудим в *главе 4*, мы можем и должны пытаться разбирать монолитные хранилища данных на части, но эта работа будет совсем не той, с которой мы хотим начать, извлекая нашу первую пару микрослужб.

Таким образом, мы смотрим на вещи через призму того, что выглядит легко и что выглядит трудно, и это занятие будет стоящим — в конце концов мы хотим отыскать несколько быстрых выигрышей! Однако мы должны помнить, что рассматриваем микрослужбы как способ достижения чего-то конкретного. Возможно, мы выясним, что "Выписка счетов-фактур", на самом деле, представляет собой простой первый шаг, но если наша цель — помочь сократить время до рынка, а функциональность "Выписка счетов-фактур" едва ли изменится, то это не будет хорошим использованием нашего времени.

Следовательно, нам нужно объединить наше представление о том, что легко и что трудно, вместе с нашим представлением о том, какие выгоды принесет декомпозиция на микрослужбы.

Комбинированная модель

Мы хотим быстрых выигрышей ради прогресса на ранней стадии, создать ощущение импульса и получить своевременную обратную связь об эффективности нашего подхода. Это подтолкнет нас к желанию выбирать для извлечения более легкие вещи. Но мы также должны получить некоторые выгоды от декомпозиции, тогда как учитывать все это в нашем мышлении?

В принципе, имеют смысл обе формы приоритизации, но нам нужен механизм для визуализации обеих форм вместе и принятия соответствующих компромиссов. Для этой цели мне нравится использовать простую структуру, как показано на рис. 2.7. Каждая кандидатная служба, выбранная для извлечения, размещается вдоль двух отображаемых осей. Ось *x* представляет значение, которое, по вашему мнению, принесет декомпозиция. Вдоль оси *y* вы упорядочиваете вещи, основываясь на их трудности.

Работая над этим процессом в группе, вы составите представление о том, что могло бы быть хорошим кандидатом для извлечения — и, как и в любой хорошей квадрантно-групповой модели, нам нравится все, что находится в правом верхнем углу,

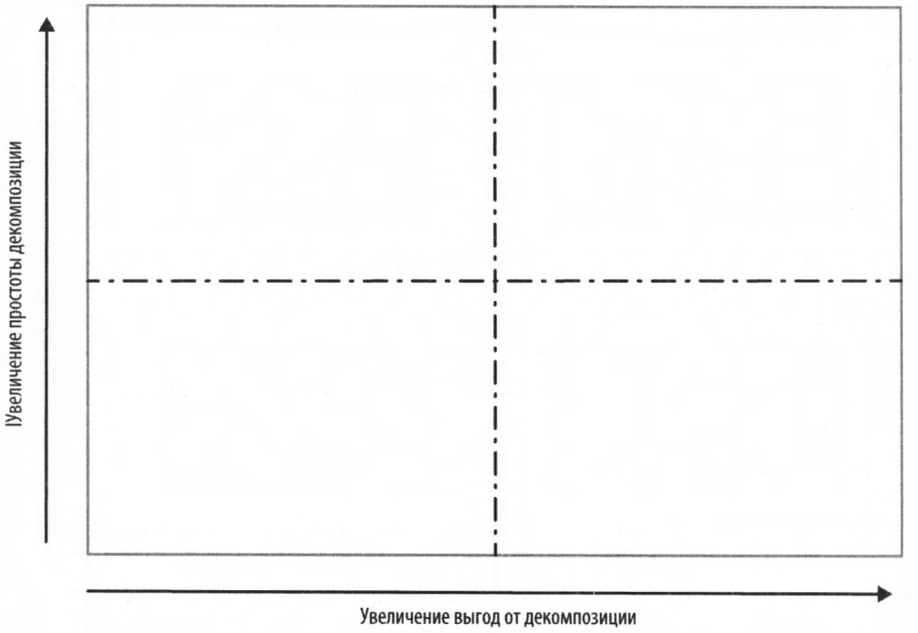


Рис. 2.7. Простая двухосевая модель для приоритизации декомпозиции на службы

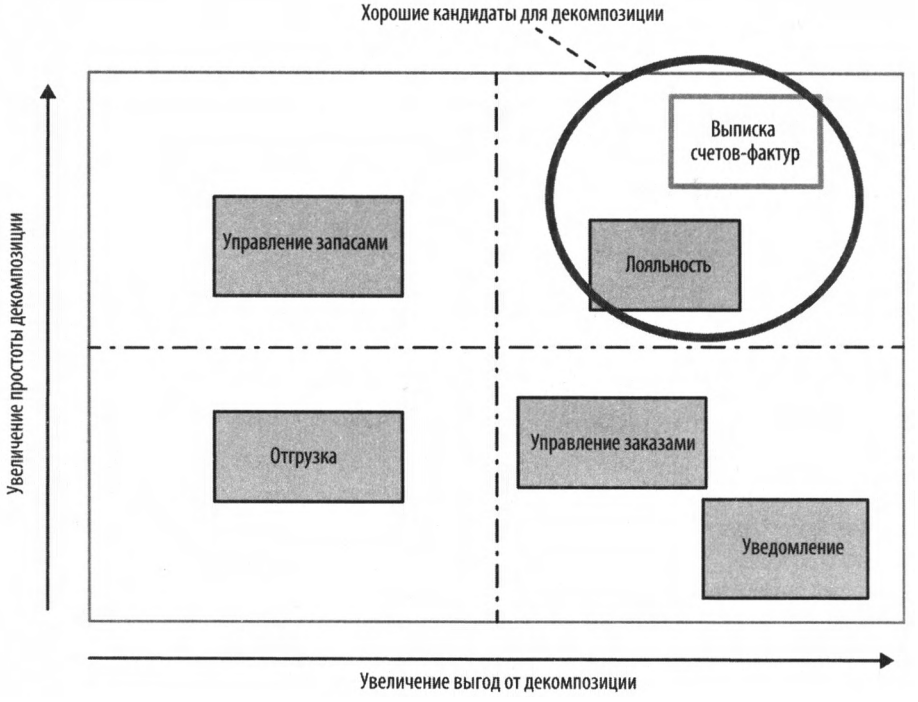


Рис. 2.8. Пример использования квадранта приоритизации

как показано на рис. 2.8. Функциональность в этом квадранте, включая "Выписку счетов-фактур", по нашему мнению, легко будет извлечь, это также принесет некоторую выгоду. Поэтому, выберите один элемент (или, возможно, два) из данной группы в качестве первых извлекаемых служб.

Когда вы начнете вносить изменения, вы научитесь большему. Некоторые вещи, которые вы считали легкими, окажутся трудными, а некоторые, казалось бы, трудные, будут легкими. И это естественно! Но это еще означает необходимость возвращаться к указанному усилию по приоритизации и заново составлять план, по мере того как вы учитесь все больше. Возможно, откалывая по кусочку, вы начинаете осознавать, что "Уведомление" на самом деле легче извлечь, чем вы думали раньше.

Реорганизация групп

После этой главы мы в основном сосредоточимся на изменениях, которые вам нужно будет вносить в свою архитектуру, и коде для осуществления успешного транзита на микрослужбы. Но, как мы уже выяснили, ключом к получению максимальной отдачи от архитектуры на основе микрослужб будет выравнивание по архитектуре и организации.

Однако вы можете оказаться в ситуации, когда, для того чтобы воспользоваться этими новыми идеями, должна измениться ваша организация. Хотя углубленное изучение организационных изменений выходит за рамки этой книги, я хочу оставить вам несколько идей, прежде чем мы погрузимся в более глубокую техническую сторону вещей.

Сдвиг в структуре

Исторически ИТ-организации структурировались вокруг стержневых компетенций. Разработчики на Java работали в одной группе с другими разработчиками на Java. Тестировщики находились в одной группе с другими тестировщиками. Все администраторы БД были в своей группе. Во время создания софта специалисты из таких групп назначались для работы над этими часто недолговечными инициативами.

Следовательно, акт создания софта требовал многочисленных эстафетных передач между группами. Бизнес-аналитик беседовал с клиентом и выяснял, чего он хочет. Затем аналитик составлял подробное требование и передавал его группе разработчиков для имплементации. Разработчик заканчивал некую работу и передавал ее группе по тестированию. Если группа по тестированию обнаруживала проблему, то работа отправлялась назад. Если все было в порядке, то работа переходила к операционной группе для развертывания.

Это разбиение на обособленные подразделения выглядит довольно знакомо. Возьмем слоеную архитектуру, которую мы обсуждали в предыдущей главе. Во время внедрения простых изменений слоеные архитектуры требуют изменения многочисленных служб. То же самое относится и к обособленным подразделениям в органи-

зации: чем больше групп необходимо вовлечь в создание или изменение части софта, тем больше времени это займет.

Эти обособленные подразделения разрушаются. Во многих организациях выделенные группы тестирования теперь ушли в прошлое. Вместо этого специалисты по тестированию становятся неотъемлемой частью групп доставки, позволяя разработчикам и тестировщикам работать теснее вместе. Движение DevOps также частично привело к тому, что многие организации отошли от централизованных операционных групп, вместо этого возлагая бо́льшую ответственность за операционные аспекты на группы доставки.

В ситуациях, когда роли этих выделенных групп были перенесены в группы доставки, роли централизованных групп сместились. От выполнения работы самостоятельно они перешли к помощи группам доставки в выполнении работы. Сюда входит встраивание специалистов в группы, создание самообслуживаемого инструментария, повышение квалификации или целый ряд других мероприятий. Их ответственность сместилась с делания к созданию возможностей.

Следовательно, мы все чаще видим более независимые, автономные группы, способные нести ответственность за бо́льшую часть цикла сквозной доставки, чем когда-либо прежде. Они сосредоточены не на конкретной технологии или деятельности, а на разных областях продукта — точно так же, как мы переключаемся с технически-ориентированных служб на службы, моделируемые вокруг вертикальных кусков бизнес-функциональности. Теперь самое важное понять, что несмотря на очевидность указанного тренда в течение многих лет, он не универсален, и такой сдвиг не является быстрой трансформацией.

Не одна мерка для всех

Мы начали эту главу с обсуждения того, как ваше решение об использовании микрослужб должно основываться на трудностях, с которыми вы сталкиваетесь, и изменениях, которые вы хотите осуществить. Такую же важность имеет внесение изменений в вашу организационную структуру. Понимание того, должна ли ваша организация измениться и как измениться, должно основываться на вашем контексте, вашей рабочей культуре и ваших людях. Вот почему простое копирование организационного дизайна у других таит в себе особую опасность.

Ранее мы очень кратко коснулись модели Spotify. Возрос интерес к тому, как Spotify выстроила свою организацию в известной статье 2012 года "Масштабирование Agile @ Spotify" (Scaling Agile @ Spotify)[14] Хенрика Книберга (Henrik Kniberg) и Андерса Иварссона (Anders Ivarsson). В той статье популяризировались понятия отрядов, орденов и гильдий, терминов, которые в нашей отрасли сейчас являются обычными (хотя и неправильно понятыми). В конечном счете, это привело к тому, что люди окрестили это "моделью Spotify", хотя данный термин никогда не использовался в Spotify.

[14] См. **http://bit.ly/2ogAz3d**.

Впоследствии, компании поспешно стали перенимать эту структуру. Но, как и в случае с микрослужбами, многие организации тяготели к модели Spotify, не задумываясь о контексте, в котором работает Spotify, об их бизнес-модели, задачах, с которыми они сталкиваются, или культуре компании. Оказывается, что организационная структура, которая хорошо работала для шведской музыкальной стриминговой компании, может не работать для инвестиционного банка. Кроме того, в оригинальной статье был показан снимок того, как Spotify работала в 2012 году, а с тех пор все изменилось. Оказывается, даже Spotify не использует "модель Spotify".

То же самое должно относиться и к вам. Черпайте вдохновение из того, что сделали другие организации, и никак иначе, но не исходите из того, что если что-то работало у кого-то другого, то оно будет работать и в вашем контексте. Как однажды выразилась Джессика Керр (Jessica Kerr) по поводу модели Spotify "Копируйте вопросы, а не ответы"[15]. Снимок организационной структуры Spotify отражает изменения, которые она провела для решения своих проблем. Копируйте это гибкое, вопрошающее отношение в том, как вы делаете вещи, и пробуйте новые вещи, но обеспечьте, чтобы изменения, которые вы применяете, коренились в понимании вашей компании, ее потребностей, ее людей и ее культуры.

Приводя конкретный пример, я вижу, как многие компании говорят своим группам доставки "Так, теперь вы должны разворачивать софт и обеспечивать поддержку 24/7". Такой подход является невероятно разрушительным и бесполезным. Иногда, большие, смелые заявления бывают отличным способом побудить все завертеться, но будьте готовы к хаосу, который он принесет. Если вы работаете в среде, где разработчики привыкли работать с 9 до 5, не будучи на связи, никогда не работали в среде технической поддержки или операций и не отличат SSH от своего локтя, то это отличный способ оттолкнуть свой персонал и потерять много людей. Если вы считаете, что для вашей организации такой шаг будет правильным, то отлично! Но говорите об этом как о стремлении, цели, которую вы хотите достичь, и объясните причину. Затем работайте со своими людьми, организуя путешествие к этой цели.

Если вы действительно хотите сделать сдвиг в сторону групп, более полно владеющих всем жизненным циклом своего софта, то вам следует понять, что навыки этих групп должны измениться. Вы можете предоставить помощь и обучение, добавить в группу новых людей (возможно, путем встраивания в группы доставки людей из текущей операционной группы). Независимо от того, какие изменения вы хотите внести, как и с нашим софтом, вы можете это осуществить в поступательном режиме.

DevOps не означает NoOps!

Существует широко распространенная путаница вокруг термина DevOps, причем некоторые люди считают, что указанный термин означает, что разработчики выполняют все операции и что люди со стороны операций не нужны. Это далеко не так. По сути, DevOps — это культурное движение, основанное на концепции разрушения барьеров между разработкой и операциями (по тестированию, развертыванию и сопровождению — *Пер.*). Вы, возможно, и не хотите, чтобы специалисты по-прежнему выступали

[15] См. **http://bit.ly/2AKTaXP**.

в этих ролях, но независимо от того, что вы хотите сделать, вы хотите содействовать общему выравниванию и пониманию между людьми, которые участвуют в доставке вашего софта, независимо от их конкретных обязанностей.

Для получения дополнительной информации я рекомендую книгу "Топологии групп" (Team Topologies)[16], в которой исследуются организационные структуры DevOps. Еще один отличный ресурс по этой теме, хотя и более широкий по охвату, — "Справочник по Devops" (The Devops Handbook)[17].

Внесение изменений

Итак, если вам не следует просто копировать чужую структуру, то с чего вы должны начинать? Во время работы с организациями, которые меняют роль групп доставки, я хотел бы начать с явного перечисления всех мероприятий и обязанностей, связанных с доставкой софта в рамках этой компании. Далее, увяжите эти мероприятия с существующей организационной структурой.

Если вы уже моделировали свой путь к производству (то, чего я большой поклонник), то вы могли бы наложить эти контуры владения на существующий вид. С другой стороны, будут хорошо работать некоторые простые вещи, такие, как на рис. 2.9. Просто возьмите заинтересованные стороны из всех вовлеченных ролей, и проведите групповой мозговой штурм всех мероприятий, которые в вашей компании входят в доставку софта.

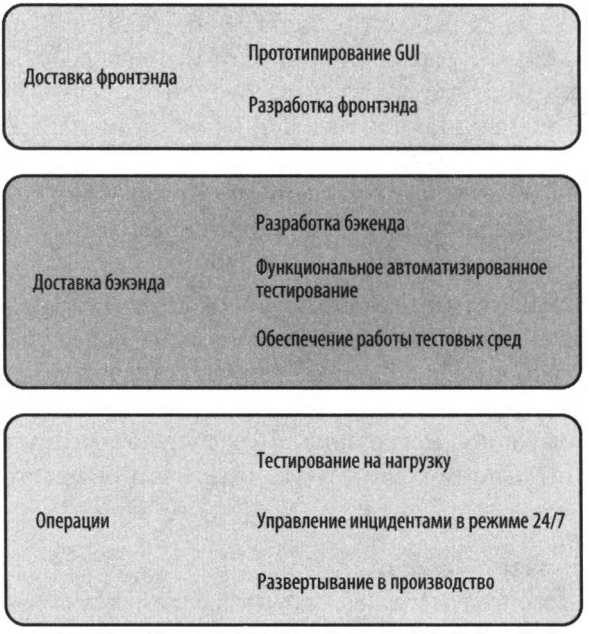


Рис. 2.9. Подмножество обязанностей по доставке и их увязка с существующими группами

[16] Мануэль Паис (Manuel Pais) и Мэтью Скелтон (Matthew Skelton), Team Topologies (IT Revolution Press, 2019).

[17] Джин Ким (Gene Kim), Джез Хамбл (Jez Humble) и Патрик Дебуа (Patrick Debois), The DevOps Handbook (IT Revolution Press, 2016).

Очень важно иметь такое понимание текущего реального (как есть) состояния, поскольку оно поможет дать каждому совместное понимание всей связанной с этим работы. Природа обособленной организации состоит в том, что вы с трудом понимаете, что делает одно обособленное подразделение, когда вы находитесь в другом обособленном подразделении. Я считаю, что это действительно помогает организациям быть честными перед самими собой относительно того, как быстро все может измениться. Скорее всего, вы также обнаружите, что не все группы равны — некоторые уже умеют многое делать сами по себе, а другие будут полностью зависеть от других групп во всем — от тестирования до развертывания.

Если вы обнаружите, что ваши группы доставки уже самостоятельно развертывают софт для тестов и тестирования со стороны пользователей, то шаг к развертыванию в производстве будет не таким большим. С другой стороны, вам все еще нужно учитывать влияние взятия на себя нижнеярусной технической поддержки (нося с собой пейджер), диагностики производственных вопросов и т. д. Эти навыки накапливаются людьми в течение многих лет работы, и ожидать, что разработчики быстро справятся с этим за одну ночь, совершенно нереально.

После того как у вас есть реальная картина, перерисуйте ваше видение того, как все должно быть в будущем, в течение некоторой разумной временной шкалы. Я нахожу, что вы захотите разведать подробно не более чем на шесть месяцев и до года вперед. Какие обязанности переходят из рук в руки? Как вы собираетесь осуществить этот транзит? Что нужно, чтобы сделать этот сдвиг? Какие новые навыки понадобятся группам? Каковы приоритеты различных изменений, которые вы хотите осуществить?

Если взять наш предыдущий пример, то на рис. 2.10 видно, что мы решили объединить обязанности групп по фронтэнду и бэкэнду. Мы также хотим, чтобы группы могли обеспечивать работу своих собственных тестовых сред. Но для этого операционная группа должна предоставить самообслуживаемую платформу, которую группа доставки будет использовать. Мы хотим, чтобы в конечном счете группа доставки брала на себя полную поддержку своего софта, и поэтому мы желаем, чтобы группы начали чувствовать себя более удовлетворенными от связанной с этим работы. Наличие собственных тестовых развертываний во владении групп доставки — хороший первый шаг. Мы также решили, что они будут обрабатывать все инциденты в течение рабочего дня, давая им возможность ускорить этот процесс в безопасной среде, где существующая операционная группа находится под рукой и будет их направлять и тренировать.

Вид крупным планом по-настоящему помогает, когда вы только приступаете к изменениям, которые хотите внести, но вам также нужно будет провести время с людьми на местах для выяснения того, осуществимы ли эти изменения, и если да, то как их осуществить. Разделив вещи между конкретными обязанностями, вы также можете подойти к этому сдвигу поступательно. Для вас концентрация внимания на устранении необходимости обеспечивать работу тестовых сред со стороны операционных групп станет правильным первым шагом.

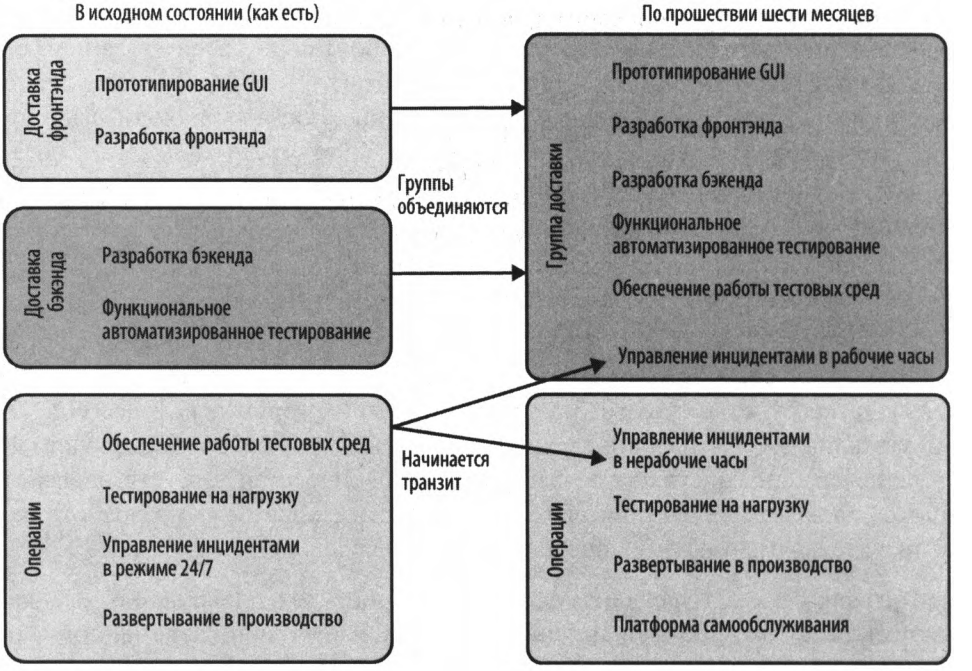


Рис. 2.10. Один из примеров того, как можно переназначить обязанности в нашей организации

Изменение навыков

В том, что касается определения необходимых людям навыков и оказания им помощи в преодолении разрыва, то я большой поклонник того, когда люди оценивают себя сами и используют это для составления более широкого понимания того, какая поддержка группе понадобится для осуществления этого изменения.

Конкретным примером этой идеи в действии является проект, в котором я участвовал во время моего пребывания в ThoughtWorks. Мы были наняты помочь газете Guardian восстановить свое онлайновое присутствие (к этому мы вернемся в следующей главе). В рамках этого проекта они должны были быстро освоить новый язык программирования и связанные с ним технологии.

В начале проекта наши объединенные группы составили список стержневых навыков, которые были важны для разработчиков Guardian. Затем каждый разработчик оценивал себя по этим критериям, ранжируя себя от 1 ("для меня это ничего не значит!") до 5 ("я мог бы написать об этом книгу"). Балл каждого разработчика был приватным, он был доступен только тому, кто их наставлял. Цель состояла не в том, чтобы каждый разработчик доводил каждый навык до "5", а наоборот в том, чтобы они сами устанавливали достигаемые ими цели.

Моя работа, как тренера, состояла в том, чтобы в случае если один из разработчиков, которых я тренировал, хотел улучшить свои навыки в Oracle, то я способствовал тому, чтобы у него была возможность этого добиться. Сюда входит обеспечение того, чтобы он работал над вариантами, в которых эта технология применялась,

рекомендации видео для просмотра, возможность участия в учебном курсе или конференции и т. д.

Вы можете использовать этот процесс для раскручивания визуального представления областей, где человек, возможно, захочет сосредоточить свое время и усилия. На рис. 2.11 мы видим такой пример, который показывает, что я действительно хочу сосредоточить свое время и энергию на развитии своего опыта в Kubernetes и Lambda, что, возможно, свидетельствует о том, что теперь я собираюсь управлять развертыванием своего собственного софта. Так же важно выделить те области, которыми вы довольны — в данном примере я чувствую, что мое кодирование на Go не является тем, на чем мне нужно сосредотачиваться прямо сейчас.

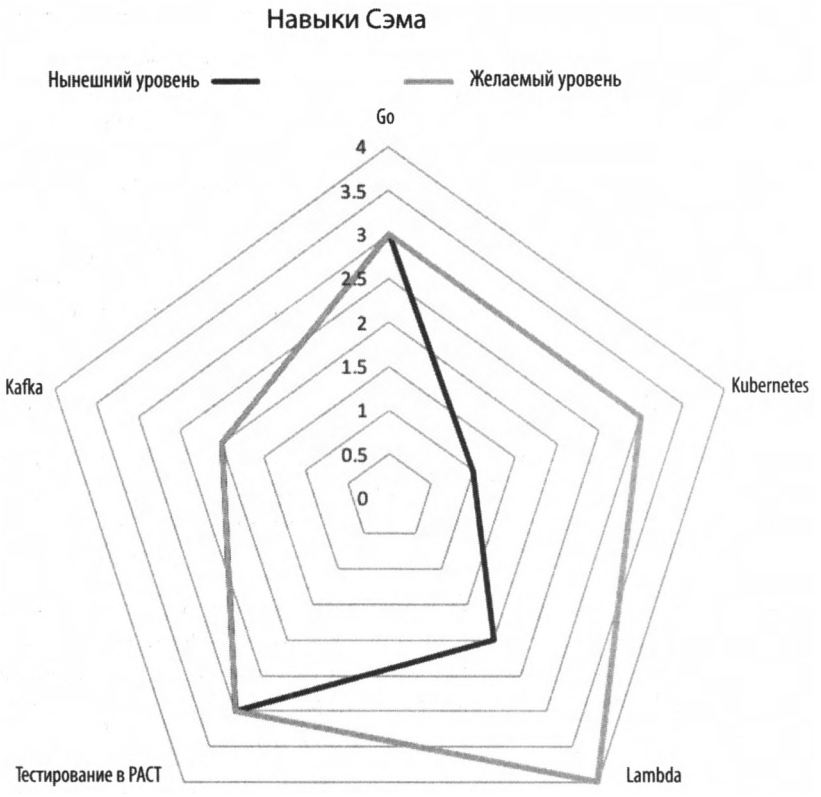


Рис. 2.11. Пример диаграммы навыков, показывающий те области, которые я хочу улучшить

Очень важно держать такого рода самооценку в тайне. Дело не в том, чтобы кто-то возвышал себя относительно кого-то другого, а в том, чтобы люди помогали направлять свое собственное развитие. Стоит их опубликовать, и вы резко измените параметры этого усилия. Внезапно люди будут беспокоиться о том, чтобы не дать себе низкую оценку, поскольку, например, это может повлиять на их аттестацию.

Хотя каждый балл является приватным, вы все равно можете его использовать для составления картины группы в целом. Возьмите анонимизированные рейтинги са-

мооценки и разработайте матрицу навыков для всей группы. Это поможет выявить пробелы, которые, возможно, потребуется устранить на более системном уровне. На рис. 2.12 показано, что хотя я могу быть доволен своим уровнем овладения платформой PACT, в целом группа желает улучшить свои навыки в этой области больше, в то время как Kafka и Kubernetes — еще одно место, которое потребует интенсивного внимания. Это, возможно, подчеркнет необходимость некоего группового обучения и оправдает более крупные вложения, такие, как проведение внутреннего учебного курса. Распространение этой совокупной картины среди вашей группы также поможет людям понять, как им принять участие в оказании помощи группе в целом с целью отыскания необходимого баланса.

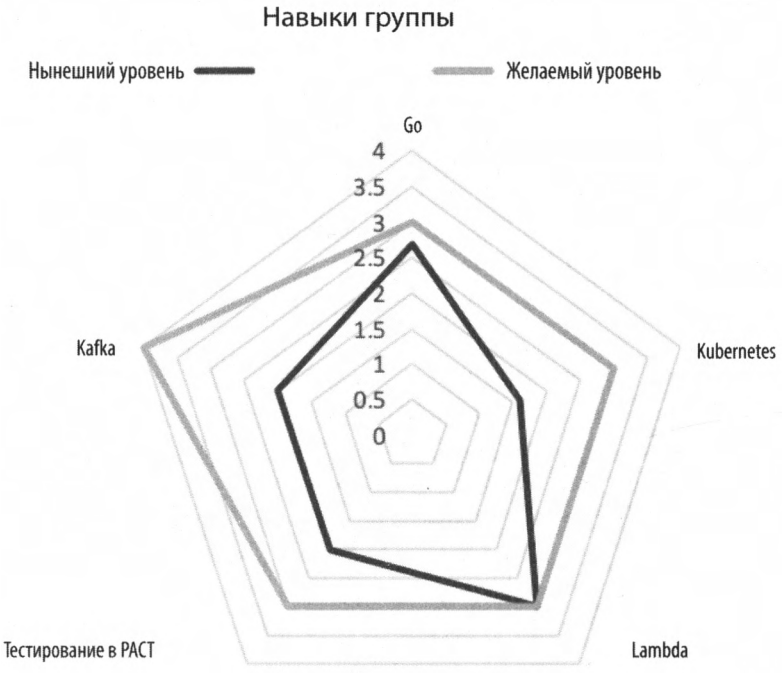


Рис. 2.12. Если смотреть в целом, то у группы есть потребность улучшить свои навыки в Kafka и Kubernetes и тестирования в PACT

Изменение набора навыков существующих членов группы, конечно, не единственный путь вперед. То, к чему мы часто стремимся, олицетворено в группе доставки, которая в целом берет на себя бо́льшую часть обязанностей. Это вовсе не означает, что каждый человек делает больше. Правильный ответ будет заключаться в привлечении в группу новых людей, обладающих необходимыми навыками. Вместо того чтобы помогать разработчикам узнавать больше о Kafka, вы можете нанять эксперта по Kafka, который присоединится к вашей группе. Это могло бы решить краткосрочную проблему, и потом у вас в группе будет эксперт, который поможет своим коллегам узнать больше и в этой области тоже.

По этой теме можно провести гораздо более детальный разведывательный анализ, но я надеюсь, что поделился в достаточной мере, чтобы дать вам возможность на-

чать работу. Прежде всего, она начинается с понимания ваших собственных людей и культуры, а также потребностей ваших пользователей. Во что бы то ни стало вдохновляйтесь примерами из других компаний, но только не удивляйтесь, если рабское копирование чужих ответов на их проблемы в итоге ничего хорошего вам не даст.

Как узнать, что транзит работает?

Мы все совершаем ошибки. Даже если вы начинаете путешествие к микрослужбам с самыми лучшими намерениями, вы должны принять как данность, что невозможно знать все и что когда-нибудь по ходу дела вы, возможно, осознаете, что все это не работает. Вопросы таковы: знаете ли вы, что он работает? Совершили ли вы ошибку?

Основываясь на желаемых целевых результатах, вы должны попытаться определить некоторые показатели, которые можно отслеживать и которые помогут вам ответить на эти вопросы. Вскоре мы разведаем некоторые примеры показателей, но я хочу воспользоваться этой возможностью, чтобы подчеркнуть тот факт, что мы говорим не только о количественных показателях. Также необходимо учитывать качественную обратную связь от людей на передовой.

Эти количественные и качественные показатели должны служить основой для продолжающегося процесса ревизии. Вам нужно установить контрольные точки, которые дадут вашей группе время на то, чтобы отрефлексировать вашу работу и уяснить, двигаетесь ли вы в правильном направлении или нет. Во время этих контрольных точек следует задавать себе вопрос не просто о том, "работает ли нечто вообще?", а о том "а не попробовать ли вместо этого что-нибудь еще?"

Давайте посмотрим, как организовать эти мероприятия, проводимые в контрольных точках, а также на некоторые примеры показателей, которые мы будем отслеживать.

Наличие регулярных контрольных точек

В качестве составной части любого транзита важно встроить в ваш процесс доставки некоторое время для паузы и рефлексии, необходимое для того, чтобы проанализировать имеющуюся информацию и определить потребность в изменении курса. В малых группах они будут проходить неформально или, возможно, будут свернуты в регулярные ретроспективные усилия. В более крупных программах работы они, возможно, должны планироваться в виде четких мероприятий на регулярной основе, возможно, объединяя руководство из различных видов деятельности на ежемесячных собраниях с целью общего обзора текущего состояния дел.

Независимо от того, как часто вы предпринимаете эти усилия, и независимо от того, насколько формально (или неформально) вы их выполняете, я предлагаю обеспечить охват следующих тем:

1. Вновь подтвердить, чего вы ожидаете от транзита на микрослужбы. Если бизнес изменил направление так, что направление, в котором вы движетесь, больше не имеет смысла, тогда остановиться!
2. Просмотреть все количественные показатели, которые у вас есть, чтобы убедиться, что вы продвигаетесь вперед.
3. Запросить качественную обратную связь — считают ли люди, что все по-прежнему получается, как и планировалось?
4. Принять решение о том, что вы собираетесь изменить в будущем, если вообще будете это делать.

Количественные показатели

Показатели, которые вы выберете для отслеживания продвижения вперед, будут зависеть от поставленных целей. Если, к примеру, вы сосредоточены на сокращении времени до рынка, то имеет смысл провести замер времени цикла, числа развертываний и частоты отказов. Если вы пытаетесь выполнить масштабирование приложения, чтобы справляться с возросшей нагрузкой, то будут разумными отчеты о последних тестах производительности.

Стоит отметить, что метрики могут оказаться опасными из-за старой пословицы "получаешь то, что измеряешь". Метрики могут быть подстроены без всякого умысла или преднамеренно. Я помню, как моя жена рассказывала мне о компании, в которой она работала, где внешний поставщик отслеживался на основе числа нарядов-заказов, которые они закрывали, и оплачивался на основе этих результатов. И что случилось? Поставщик закрывал наряды-заказы, даже если задача не была решена, приводя к тому, что вместо них люди открывали новые наряды-заказы.

Другие показатели бывает трудно изменить в течение короткого периода времени. Я был бы удивлен, если вы увидели значительное улучшение времени цикла в рамках миграции на микрослужбы в первые несколько месяцев; на самом деле, я, вероятно, ожидал бы, что оно вообще ухудшится. Внесение изменений в порядок работы часто негативно сказывается на производительности в краткосрочной перспективе, пока группа быстро осваивает новый способ работы. Это еще одна причина, почему так важно делать небольшие поступательные шаги: чем меньше изменение, тем меньше будут потенциальные негативные последствия, которые вы увидите, и тем быстрее вы их устраните, когда они произойдут.

Качественные показатели

> "...Программно-информационное обеспечение состоит из чувств".
>
> – *Астрид Аткинсон* (@shinynew_oz) *(Astrid Atkinson)*

Что бы ни показывали нам наши данные, именно люди строят софт, и важно, чтобы обратная связь с их стороны была включена в измерение успеха. Получают ли они удовольствие от этого процесса? Чувствуют ли они, что у них стало больше сил? Или они чувствуют себя подавленными? Получают ли они поддержку, необходимую для того, чтобы взять на себя новые обязанности или освоить новые навыки?

Когда вы докладываете о каких-либо перечнях показателей высшему руководству для такого рода транзитов[18], вы должны включать проверку настроений, исходящих из вашей группы. Если транзит им нравится, то все отлично! Если же это не так, то вам, возможно, придется что-то с этим делать. Игнорирование мнения людей в пользу количественных показателей — отличный способ попасть в большую беду.

Избегать эффекта понесенных расходов

Вы должны отдавать себе полный отчет об эффекте понесенных расходов, и наличие ревизионного процесса отчасти подстегивает к тому, чтобы оставаться честным и, надо надеяться, помогать вам избегать этого явления. Эффект понесенных расходов возникает, когда люди настолько вкладываются в предыдущий подход к чему-то, что, даже если факты показывают, что этот подход не работает, они все равно продолжают работать в том же направлении. Иногда мы оправдываемся перед собой, говоря, что "в любую минуту все может измениться!" В других случаях внутри нашей организации мы, возможно, приложили столько политического капитала на осуществление перемен, что сейчас просто не можем дать задний ход. В любом случае, можно, без всякого сомнения, утверждать, что эффект понесенных расходов связан с эмоциональными вложениями: мы настолько "купились" на старый способ мышления, что просто не можем от него отказаться.

По моему опыту, чем выше ставка и громче сопровождающие ее "фанфары", тем труднее отступить, когда все идет не так. Эффект понесенных расходов[19] также именуемый когнитивным искажением Конкорд, названным в честь провалившегося проекта, поддержанного британским и французским правительствами, строительства сверхзвукового пассажирского самолета с огромными расходами. Несмотря на все факты, свидетельствующие о том, что указанный проект никогда не принесет финансовой отдачи, в него закачивалось все больше и больше денег. Какими бы ни были инженерные успехи, которые, возможно, пришли из Конкорда, он никогда не работал как самолет, подходящий для коммерческих пассажирских рейсов.

Если вы сделаете каждый шаг малым, то вам будет проще избежать ловушек, связанных с эффектом понесенных расходов, и изменить направление движения. Используйте рассмотренный ранее механизм контрольных точек для того, чтобы рефлексировать все происходящее. Вам не нужно отступать или менять курс при первых признаках неприятностей, но игнорировать подтверждающие свидетельства, собираемые вами относительно успеха (или неуспеха) изменения, которое вы пытаетесь осуществить, возможно, будет глупее, чем не собирать никаких свидетельств вообще.

---

[18] Да, это действительно произошло. С Kubernetes не все веселье и игры....

[19] Эффект понесенных расходов (sunk cost fallacy) — это широко известное когнитивное искажение, дословно переводимое как "ошибочность утопленных затрат", когда психика человека отказывается признать и зафиксировать понесенные затраты, "списать" их и двигаться дальше, вместо этого побуждая "вбухивать" все больше и больше — *Пер.*

Оставаясь открытым для новых подходов

Как я надеюсь, для вас не будет сюрпризом, если вы дошли до этой части главы, что в процесс разбиения монолитной системы на куски задействованы многочисленные переменные и многочисленные пути, которые мы могли бы выбрать. Уверенным можно быть только в одном — не все пойдет гладко, и вам нужно оставаться открытым для отката внесенных вами изменений, пробуя новые вещи или иногда просто давая вещам немного утрястись, чтобы увидеть, какое влияние они оказывают.

Если вы пытаетесь освоить культуру постоянного совершенствования, всегда апробируя что-то новое, тогда смена направления, когда это необходимо, становится гораздо естественнее. Если вы концентрируете концепцию изменений или улучшений процесса в отдельных трудовых потоках, вместо того чтобы встраивать ее во все, что вы делаете, то вы рискуете увидеть изменения как разовые транзакционные мероприятия. Как только работа будет сделана, на этом все! Для нас больше никаких перемен! При таком образе мыслей через несколько лет вы окажетесь позади всех своих конкурентов и с другой горой, на которую еще предстоит подняться.

Резюме

В этой главе мы охватили много вопросов. Мы рассмотрели причину, почему вы, возможно, захотите внедрить архитектуру, основанную на микрослужбах, и как это решение влияет на то, как вы приоритизируете свое время. Мы рассмотрели ключевые вопросы, которые группы должны задавать себе, принимая решение о том, подходят ли им микрослужбы или нет, и эти вопросы повторяются из раза в раз:

- Чего вы надеетесь достичь?
- Думали ли вы об альтернативах использованию микрослужб?
- Как узнать, что транзит работает?

В дополнение к этому нельзя переоценить важность применения поступательного подхода к извлечению микрослужб. Ошибки неизбежны, поэтому, принимая их как данность, вы должны стремиться совершать малые ошибки вместо больших. Разложение транзита по направлению к архитектуре на основе миксрослужб на небольшие поступательные шаги приведет к тому, что совершаемые нами ошибки окажутся малыми и от них будет легче избавиться.

Большинство из нас также работают над системами, которые имеют реальных клиентов. Мы не можем позволить себе тратить месяцы или годы на переписывание нашего приложения методом "Большого взрыва", позволяя существующему приложению, которым пользуются наши клиенты, "простаивать под паром". Целью должно быть поступательное создание новых микрослужб и их развертывание в рамках вашего производственного решения, благодаря чему вы сможете начать учиться на опыте и получать выгоды как можно раньше.

Я предельно ясен по поводу идеи о том, что во время извлечения функциональности в новую службу работа не будет завершена до тех пор, пока она не окажется

в производстве и не будет использоваться. В ходе процесса доведения ваших первых нескольких служб до использования их клиентами вы узнаете много полезного. На ранней стадии это должно быть в центре вашего внимания.

Все это означает, что нам нужно разработать серию методов, которые позволят создавать новые микрослужбы и интегрировать их с нашим (надо надеяться) сжимающимся монолитом, а затем отправлять их в производство. Дальше мы займемся шаблонами, которые показывают то, как делать эту работу, продолжая поддерживать свою систему в рабочем состоянии, обслуживать своих клиентов и интегрировать новую функциональность.

Разложение монолита

В *главе 2* мы провели разведывательный анализ того, как продумать миграцию на архитектуру, основанную на микрослужбах. Если конкретнее, мы разведали вопрос, была ли вообще эта идея хорошей, и если да, то как нужно поступать с точки зрения внедрения своей новой архитектуры и обеспечения движения в правильном направлении.

Мы обсудили вопрос о том, как выглядит хорошая служба и почему меньшие по объему службы будут для нас лучше. Но как справиться с тем фактом, что у нас уже имеется большое число приложений, которые не следуют этим шаблонам? Как выполнить декомпозицию этих монолитных приложений, не прибегая к переписыванию методом "Большого взрыва"?

В этой главе мы разведаем различные шаблоны миграции, а также дадим советы, которые помогут вам внедрить архитектуру, основанную на микрослужбах. Мы рассмотрим шаблоны, которые будут работать для поставляемого черно-ящичного софта, унаследованных систем или монолитов, которые вы планируете продолжать поддерживать и развивать. Однако, для того чтобы поступательное внедрение работало как надо, нам необходимо обеспечить продолжение работы с существующим монолитным софтом и его использование клиентами.

Помните, что мы хотим сделать нашу миграцию поступательной. Нам нужно мигрировать на архитектуру, основанную на микрослужбах, малыми шагами, что позволит нам обучаться и при необходимости менять свое мнение по ходу движения.

Изменять монолит или не изменять?

Среди первых вопросов, о которых вам необходимо подумать в рамках своей миграции, будет вопрос о планировании (или возможности) внесения изменений в существующий монолит.

Если у вас есть возможность изменить существующую систему, то это даст вам наибольшую гибкость с точки зрения различных шаблонов, имеющихся в вашем распоряжении. Однако в некоторых ситуациях возникает жесткое ограничение, лишающее вас этой возможности. Существующая система бывает продуктом поставщика, исходного кода которого у вас нет, или же она была написана с привлечением технологии, навыков в которой у вас больше нет.

Кроме того, могут иметься более мягкие мотивы, которые отвлекают вас от изменения существующей системы. Вполне возможно, что нынешний монолит находится в таком плохом состоянии, что стоимость изменений слишком высока, и, как результат, вы захотите сократить свои потери и начать заново (хотя, как я уже подробно описывал ранее, я беспокоюсь, что люди слишком легко приходят к этому выводу). Еще одна возможность заключается в том, что над самим монолитом работают многие другие люди, и вы беспокоитесь о том, чтобы не встать у них на пути. Некоторые шаблоны, такие, как шаблон "Ветвление по абстракции", который мы вскоре разведаем, эти проблемы смягчают, но вы все равно, возможно, сочтете, что влияние на других оказывается слишком большим.

В одной памятной ситуации я работал с несколькими коллегами, помогая им промасштабировать вычислительно тяжелую систему. Опорные вычисления выполнялись с помощью библиотеки C, которую нам дали. Наша задача состояла в том, чтобы собрать различные входные данные, передать их в библиотеку, а также получить и сохранить результаты. Сама библиотека была полна проблем. Утечки памяти и ужасно неэффективный дизайн API были лишь двумя главенствующими причинами проблем. В течение многих месяцев мы просили дать нам исходный код библиотеки, чтобы иметь возможность исправить эти проблемы, но нам отказали.

Много лет спустя я встретился со спонсором проекта и спросил, почему он тогда не дал нам изменить базовую библиотеку. Именно в этот момент спонсор, наконец-таки, признался, что он потерял исходный код, но был слишком смущен, и поэтому ничего нам не сказал! Не давайте этому случиться с вами.

Поэтому, надеюсь, мы находимся в положении, когда мы можем работать с существующей монолитной кодовой базой и в состоянии ее изменить. Но если мы не можем, то значит ли это, что мы в тупике? Совсем наоборот — здесь нам помогут несколько шаблонов. Вскоре мы займемся рассмотрением некоторых из них.

Вырезать, скопировать или реимплементировать?

Даже если в самом начале миграции функциональности на новые микрослужбы у вас есть доступ к существующему коду в монолите, не всегда ясно, что делать с этим кодом. Следует ли нам перенести код как есть или же реимплементировать функциональность?

Если существующая монолитная кодовая база достаточно хорошо факторизована, то вы сэкономите значительное время, перенеся сам код. Главное здесь — понять, что мы хотим именно *скопировать* код из монолита и, по крайней мере, на данном этапе не хотим удалять эту функциональность из самого монолита. Почему? Потому что, оставляя функциональность в монолите на некоторое время, вы получаете больше возможностей. Это даст нам точку отката или же возможность выполнять обе имплементации параллельно. И дальше по ходу дела, как только вы будете довольны тем, что миграция прошла успешно, вы удалите эту функциональность из монолита.

Рефакторизация монолита

Я заметил, что часто самым большим препятствием для использования существующего кода из монолита в ваших новых микрослужбах является то, что существующие кодовые базы традиционно не организованы вокруг понятий бизнес-домена — более заметны технические категоризации (например, подумайте обо всех пакетных именах "Модель-Вид-Контроллер", которые вы встречали). Когда вы пытаетесь перенести функциональность, относящуюся к бизнес-домену, это бывает трудно сделать: существующая кодовая база не соответствует этой категоризации, поэтому даже отыскать код, который вы пытаетесь перенести, бывает проблематично!

Если вы действительно встали на путь реорганизации существующего монолита вдоль контуров бизнес-домена, то я настоятельно рекомендую книгу "Эффективная работа с унаследованным кодом" Майкла Фитерса (Michael Feathers)[1]. В своей книге Майкл определяет понятие стыка (seam, шов), т. е. места, где можно изменить программу без необходимости редактировать существующее поведение. По сути, вы определяете стык вокруг фрагмента кода, который хотите изменить, работаете над новой имплементацией стыка и подставляете ее после внесения изменений. Автор показывает методику безопасной работы со стыками как способ очистки кодовых баз.

Хотя в целом концепция стыков Майкла Фитерса применима во многих областях, она очень хорошо вписывается в ограниченные контексты, которых мы касались в *главе 1*. Поэтому, хотя книга "Эффективная работа с унаследованным кодом" не ссылается непосредственно на концепции доменно-обусловленного дизайна, вы можете воспользоваться методологией, описанной в той книге, для организации своего кода в соответствии с этими принципами.

Модульный монолит?

После того как ваша существующая кодовая база начала обретать смысл, стоит подумать о следующем очевидном шаге — взять ваши только что выявленные стыки и начать извлекать их как отдельные модули, превращая ваш монолит в модульный монолит. У вас по-прежнему одна единица развертывания, но эта развернутая единица состоит из многочисленных статически связанных модулей. Точная природа этих модулей зависит от вашего опорного стека технологий — для Java мой модульный монолит будет состоять из нескольких файлов JAR; для приложения Ruby это будет коллекция из gem-ов языка Ruby.

Как мы кратко упомянули в начале книги, наличие монолита, разложенного на модули, которые развиваются независимо, приносит много выгод при одновременном обходе многих трудностей архитектуры, основанной на микрослужбах, и является наиболее благоприятной зоной для многих организаций. Я беседовал с несколькими группами, начавшими дезинтегрировать свой монолит на модульный монолит,

[1] См. Эффективная работа с унаследованным кодом (Working Effectively with Legacy Code, Michael Feathers, Prentice Hall, 2004).

о перспективе в конечном счете перейти на архитектуру, основанную на микрослужбах, и выяснил лишь то, что модульный монолит решил большинство их проблем!

Поступательные переписывания

Мое общее стремление — всегда сначала пытаться спасти существующую кодовую базу, прежде чем прибегнуть к простой реимплементации функциональности, и совет, который я дал в своей предыдущей книге "Создание микросервисов", был именно в этом плане. Иногда группы считают, что они получают от этой работы выгоды, достаточные для того, чтобы вообще не нуждаться в микрослужбах!

Однако должен признаться, на практике я нахожу, что очень немногие группы используют подход, предусматривающий рефакторизацию своего монолита, в качестве предшественника транзита на микрослужбы. Вместо этого более общепринятый подход, похоже, состоит в том, чтобы, после того как группы выявили обязанности только что созданной микрослужбы, они выполняли новую имплементацию этой функциональности в "чистой комнате".

Но разве мы не рискуем повторить проблемы, ассоциированные с переписываниями методом "Большого взрыва", если начнем реимплементировать нашу функциональность? Ключ находится в обеспечении переписывания только малых кусков функциональности за один раз и доставке этой переработанной функциональности своим клиентам на регулярной основе. Если работа по реимплементации поведения службы занимает несколько дней или недель, то это, вероятно, будет нормально. Если же временные рамки начнут выглядеть более похожими на несколько месяцев, то я подвергну свой подход переэкзаменовке.

Шаблоны миграции

Я встречал применение многих методов, которые использовались в рамках миграции на микрослужбы. В оставшейся части главы мы разведаем эти шаблоны, рассмотрев, где они могут быть полезны и как их можно имплементировать. Помните, что, как и все шаблоны, они не являются универсально "хорошими" идеями. По каждому из них я попытался привести информацию в достаточной мере, для того чтобы помочь вам понять, имеют ли они смысл в вашем контексте.

Убедитесь, что вы понимаете плюсы и минусы каждого из этих шаблонов. Они не всегда являются "правильным" способом делать вещи.

Мы начнем с рассмотрения методов, позволяющих вам мигрировать и интегрироваться с монолитом; в первую очередь мы займемся вопросом о том, где располагается прикладной код. Для начала, однако, мы рассмотрим один из самых полезных и часто применяемых методов: приложение "Фикус-удавка".

Шаблон: приложение "Фикус-удавка"

Метод, который часто используется при переписывании системы, называется приложением "Фикус-удавка" (strangler fig app). Мартин Фаулер был первым, кто выявил этот шаблон, находясь под впечатлением особого вида фикуса, который поселяется на верхних ветвях деревьев. Со временем фикус опускается к земле и пускает корни, постепенно обволакивая первоначальное дерево. Существующее дерево первоначально становится опорной структурой для нового фикуса, и если он дойдет до заключительных стадий, то вы увидите, как первоначальное дерево умирает и трухлявеет, оставляя на своем месте лишь новый, теперь уже самоподдерживающийся фикус.

В контексте софта параллель здесь состоит в том, чтобы наша новая система изначально поддерживалась существующей системой и обертывала ее. Идея состоит в том, что старое и новое могут сосуществовать, давая новой системе время вырасти и потенциально полностью заменить собой старую систему. Ключевая выгода от этого шаблона, как мы вскоре увидим, заключается в том, что он поддерживает нашу цель обеспечения поступательной миграции на новую систему. Более того, он дает нам возможность ставить на паузу и даже полностью останавливать миграцию, по-прежнему используя преимущества новой системы, поставленной к этому времени.

Как мы вскоре увидим, когда мы имплементируем эту идею для нашего софта, то стремимся не только делать поступательные шаги к нашей новой архитектуре приложения, но и обеспечивать, чтобы каждый шаг был легко обратимым, уменьшая риск на каждом шаге.

Как он работает

В то время как шаблон "Фикус-удавка" обычно использовался для мигрирования из одной монолитной системы в другую, мы будем искать возможность миграции из монолита на серию микрослужб. Сюда будет входить фактическое копирование кода из монолита (если это возможно), или же реимплементация затрагиваемой функциональности. В дополнение к этому, если затрагиваемая функциональность требует поддержки постоянства состояния, то нужно уделить внимание тому, как это состояние будет мигрировано в новую службу и, возможно, обратно. В *главе 4* мы рассмотрим аспекты, связанные с данными.

Имплементация шаблона "Фикус-удавка" основана на трех шагах, как показано на рис. 3.1. Прежде всего, определить части существующей системы, которые вы хотите мигрировать. Вам необходимо будет принять решение о том, какими частями системы следует заняться в первую очередь, используя что-то вроде компромиссных мероприятий, которые мы обсуждали в *главе 2*. Затем вам нужно имплементировать эту функциональность в своей новой микрослужбе. Когда ваша новая имплементация будет готова, вы должны иметь возможность перенаправлять вызовы из монолита в вашу совершенно новую микрослужбу.

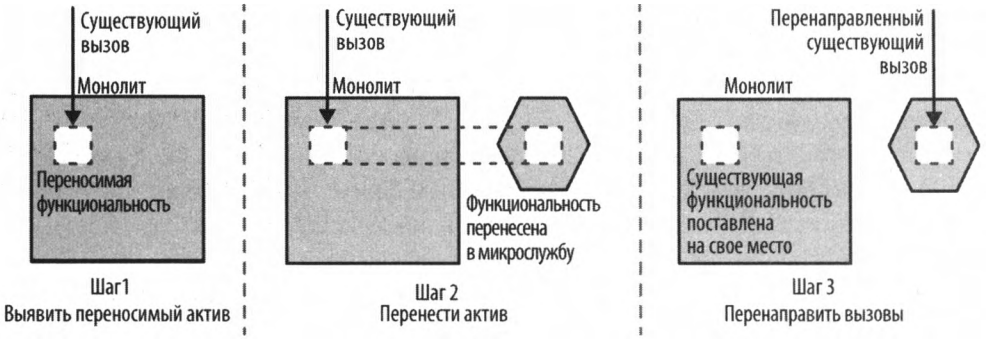


Рис. 3.1. Общий вид шаблона "Фикус-удавка"

Стоит отметить, что, до тех пор пока вызов перенесенной функциональности не будет перенаправлен, новая функциональность технически не будет находиться "в прямом эфире" — даже если она развернута в рабочей среде. Это означает, что вы можете не торопиться с приведением данной функциональности в соответствующий вид, работая с имплементацией этой функциональности в течение некоторого периода времени. Вы можете вложить эти изменения в производственную среду с осознанием того, что она еще не используется, позволяя нам получать удовлетворение от развертывания вашей новой службы и управления ею. После того, как ваша новая служба будет имплементировать ту же самую эквивалентную функциональность, что и в вашем монолите, вы сможете подумать о применении такого шаблона, как "Параллельное выполнение" (который мы вскоре рассмотрим), который даст вам уверенность в том, что новая функциональность работает так, как задумано.

Важно разделять понятия развертывания и релиза. Просто потому что программно-информационное обеспечение развертывается в заданной среде, не означает, что оно используется клиентами фактически. Рассматривая эти две вещи как отдельные понятия, вы даете возможность проверить свой софт в конечной производственной среде до его использования, что позволяет избежать риска развертывания нового софта. Шаблоны, подобные "Фикусу-удавке", "Параллельному выполнению" и "Канареечному релизу", относятся к числу тех шаблонов, которые учитывают тот факт, что развертывание и релиз представляют собой отдельные мероприятия.

Ключевой момент в этом подходе с использованием приложения-удавки заключается не только в том, что мы мигрируем новую функциональность в новую систему поступательно, но и в том, что при необходимости мы можем очень легко откатить это изменение назад. Помните, что все мы совершаем ошибки, поэтому нам нужны методы, позволяющие не только совершать ошибки как можно дешевле (отсюда и много мелких шагов), но и быстро их исправлять.

Если извлекаемая функциональность также используется другими функциональностями внутри монолита, то вам также нужно изменить способ выполнения ее вызовов. Мы рассмотрим несколько таких методов далее в этой главе.

Где его использовать

Шаблон "Фикус-удавка" позволяет переносить функциональность на новую архитектуру, основанную на службах, не касаясь и не внося никаких изменений в существующую систему. Он выгоден, когда над существующим монолитом как таковым работают другие люди, т. к. он помогает уменьшить конкуренцию. Он также очень полезен, когда монолит фактически является черно-ящичной системой — такой, как сторонний софт или служба SaaS.

От случая к случаю можно извлекать весь сквозной кусок функциональности целиком, как показано на рис. 3.2. Это значительно упрощает извлечение, не считая вопросов, связанных с данными, которые мы рассмотрим в этой книге позже.

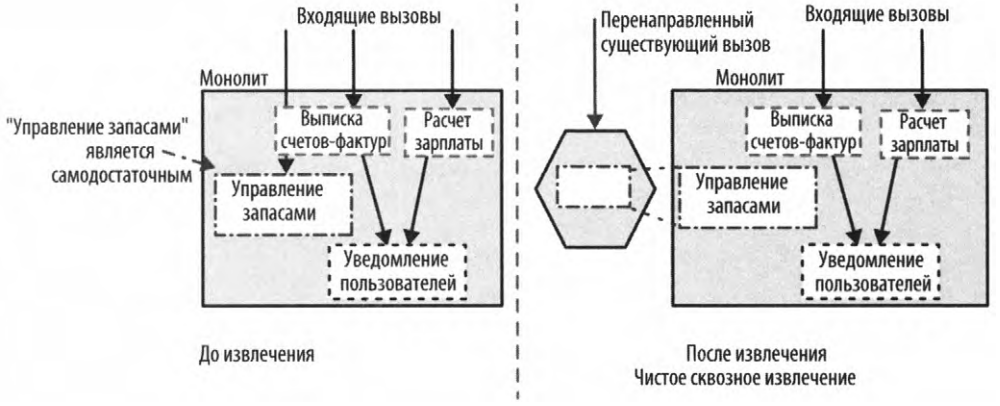


Рис. 3.2. Прямолинейная сквозная абстракция функциональности "Управления запасами"

Для того чтобы выполнить чистое сквозное ("конец-в-конец") извлечение, как тут, вы, возможно, будете стремиться извлекать более крупные куски функциональности, чтобы упростить этот процесс. Это приведет к запутанному уравновешивающему акту: извлекая более крупные куски функциональности, вы берете на себя больше работы, но упрощаете некоторые трудности вашей интеграции.

Если же вы хотите откусить кусок поменьше, то, возможно, вам придется подумать о более "мелких" извлечениях, как показано на рис. 3.3. Здесь мы извлекаем функциональность "Расчета заработной платы", несмотря на то что она использует еще одну функциональность, которая остается внутри монолита, — в данном примере это возможность отправки "Уведомлений пользователям".

Вместо того чтобы также реимплементировать функциональность "Уведомлений пользователей", мы выставляем эту функциональность наружу из монолита нашей новой микрослужбе, что, очевидно, потребует изменений в самом монолите.

Однако, для того чтобы "удавка" работала как надо, необходимо, чтобы у вас была возможность четко отображать входящий вызов интересующей вас функциональности на актив, который вы хотите перенести. Например, на рис. 3.4 в идеале мы хотели бы вынести возможность отправки "Уведомлений пользователю" в новую службу. Однако уведомления запускаются в результате многочисленных входящих

вызовов существующего монолита. Следовательно, мы не можем четко перенаправлять вызовы извне самой системы. Вместо этого нам нужно бы рассмотреть метод, подобный описанному в разделе "Шаблон: ветвление по абстракции".

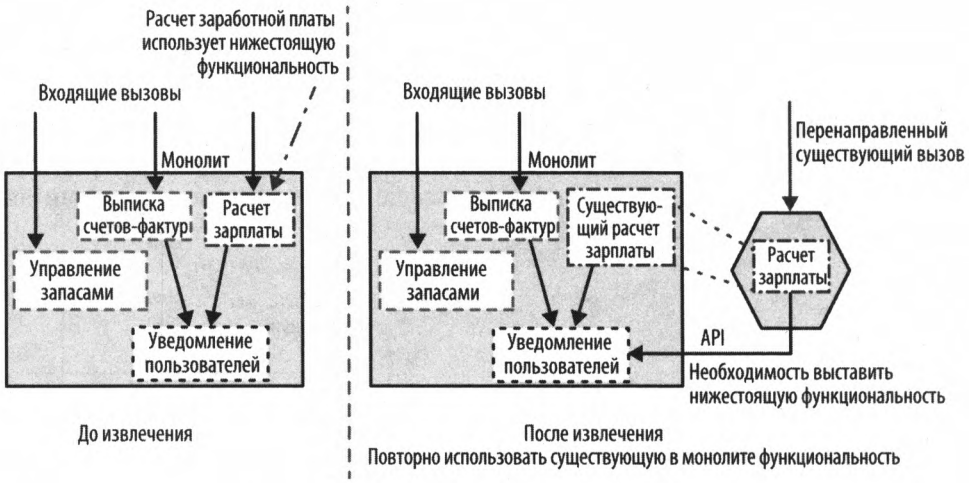


Рис. 3.3. Извлечение функциональности, которая по-прежнему нуждается в использовании монолита

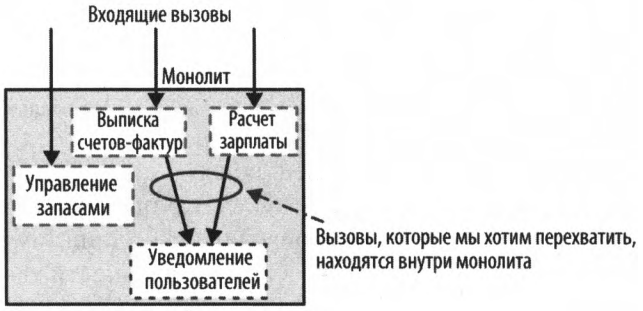


Рис. 3.4. Шаблон "Фикус-удавка" работает не слишком хорошо, когда функциональность, которую нужно перенести, находится глубже внутри существующей системы

Вам также нужно будет учесть природу вызовов, осуществляемых внутрь существующей системы. Как мы вскоре разведаем, такой протокол, как HTTP, хорошо поддается перенаправлению.

В сам HTTP встроены концепции прозрачного перенаправления, и прокси-селекторы используются для четкого понимания природы входящего запроса и его соответствующей переадресации. Другие типы протоколов, такие, как некоторые RPC, поддаются перенаправлению хуже. Чем больше работы нужно выполнить на прокси-слое, для того чтобы понять и потенциально трансформировать входящий вызов, тем менее жизнеспособным становится этот вариант.

Несмотря на эти ограничения, приложение "Фикус-удавка" снова и снова доказывает, что это очень полезный метод миграции. Учитывая легкое прикосновение и

легкий подход к работе с поступательными изменениями, нередко он является моим первым портом захода во время разведывания того, как мигрировать систему.

Пример: обратный прокси-селектор HTTP

Протокол HTTP имеет несколько интересных возможностей, среди которых очень легкий перехват и перенаправление таким образом, который может быть сделан прозрачным для вызывающей системы. Это означает, что существующий монолит с HTTP-интерфейсом поддается миграции с помощью шаблона "Фикус-удавка".

На рис. 3.5 мы видим существующую монолитную систему, которая выставляет наружу HTTP-интерфейс. Это приложение может быть "безголовым", или же HTTP-интерфейс может фактически вызываться вышестоящим пользовательским интерфейсом. В любом случае, цель одинакова: вставить обратный прокси-селектор HTTP[2] между вышестоящими вызовами и нижестоящим монолитом.

Рис. 3.5. Простой общий вид HTTP-обусловленного монолита до имплементации "удавки"

Шаг 1: вставить прокси-селектор

Если у вас еще нет подходящего прокси-селектора HTTP, который вы можете использовать много раз, то я предлагаю сначала установить его на свое место, как показано на рис. 3.6. На этом первом шаге прокси-селектор просто позволит любым вызовам проходить насквозь без изменений.

Этот шаг позволит вам оценить влияние вставки дополнительного сетевого "прыжка" между вышестоящими вызовами и нисходящим монолитом, выстроить любой

[2] Для справки: обратный прокси-селектор (reverse proxy) — это селектор, который находится перед веб-серверами и перенаправляет запросы клиента (например, веб-браузера) на эти веб-серверы. Обратные прокси обычно реализуются для повышения безопасности, производительности и надежности — *Пер.*

необходимый мониторинг вашего нового компонента и, в сущности, остаться с ним на некоторое время. С точки зрения задержки, мы добавим сетевой "прыжок" и процесс в процессном пути всех вызовов. С приличным прокси-селектором и сетью можно ожидать минимального влияния на задержку (возможно, порядка нескольких миллисекунд), но если это окажется не так, то у вас есть шанс остановиться и произвести расследование проблемы, прежде чем идти дальше.

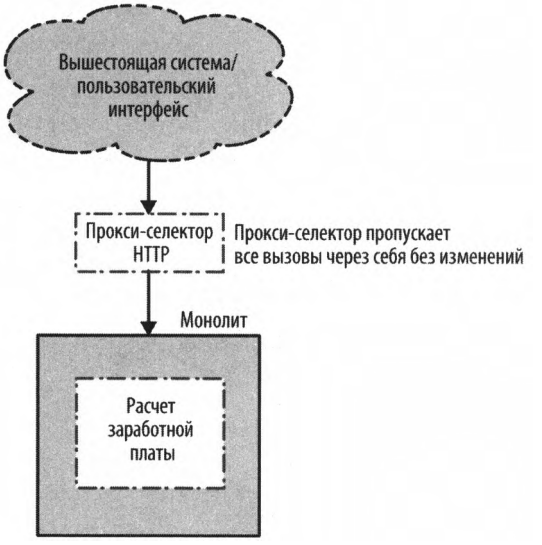


Рис. 3.6. Шаг 1: вставка прокси-селектора между монолитом и вышестоящей системой

Если перед вашим монолитом уже есть существующий прокси-селектор, то вы можете пропустить этот шаг — хотя убедитесь, что вы понимаете, как этот прокси-селектор можно реконфигурировать для перенаправления вызовов впоследствии. Я предлагаю, по крайней мере, поэкспериментировать с перенаправлением, убедиться, что все работает, как задумано, а не откладывать данный шаг на более поздний срок. Было бы неприятным сюрпризом обнаружить, что это невозможно, прямо перед тем как вы запланировали отправить свою новую службу в "прямой эфир"!

Шаг 2: мигрировать функциональность

После установки нашего прокси-селектора на свое место можно начать извлечение новой микрослужбы, как показано на рис. 3.7.

Сам этот шаг можно разбить на несколько этапов. Прежде всего, привести базовую службу в рабочее состояние без имплементации какой-либо функциональности. Ваша служба должна будет принимать вызовы соответствующей функциональности, но на этом этапе можно просто возвращать код ошибки `501 Not Implemented` (Не имплементировано). Даже на этом шаге я бы развернул службу в производственной среде. Это позволит вам освоиться с процессом развертывания в производстве и протестировать службу прямо на месте. В этой точке ваша новая служба не выпус-

кается в производственную среду, т. к. вы еще не перенаправили существующие вышестоящие вызовы. Фактически, мы отделяем шаг развертывания софта от релиза софта — распространенный метод релиза, к которому мы вернемся позже.

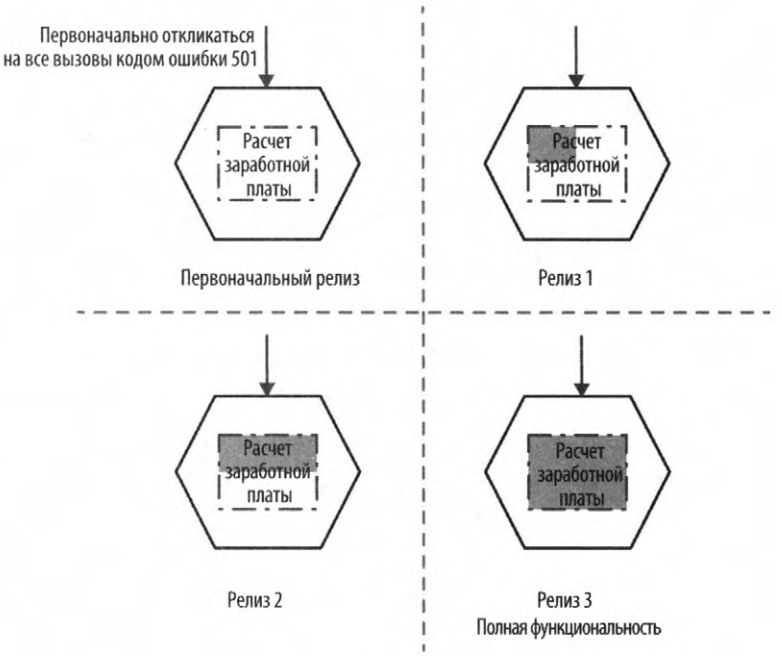


Рис. 3.7. Шаг 2: поступательная имплементация переносимой функциональности

Шаг 3: перенаправить вызовы

Только после завершения переноса всей функциональности, вы переконфигурируете прокси-селектор с целью перенаправления вызовов, как мы видим на рис. 3.8. Если по какой-либо причине это не получается, то вы можете выставить перенаправление в прежнее состояние — для большинства прокси-селекторов это очень быстрый и легкий процесс, дающий вам быстрый откат.

Возможно, вы решите имплементировать перенаправление, применив что-то вроде реле функции (флажка с двумя состояниями, тогла), способного проявить состояние вашей желаемой конфигурации гораздо отчетливее. Использование прокси-селектора для перенаправления вызовов также отличное место, для того чтобы подумать о поступательном внедрении новой функциональности посредством канареечного релиза или даже о полномасштабном параллельном выполнении, еще одном шаблоне, который мы обсудим в этой главе.

Данные?

Мы до сих пор еще не говорили о данных. Глядя на рис. 3.8, представьте, что произойдет, если нашей новой службе "Расчета заработной платы" потребуется обра-

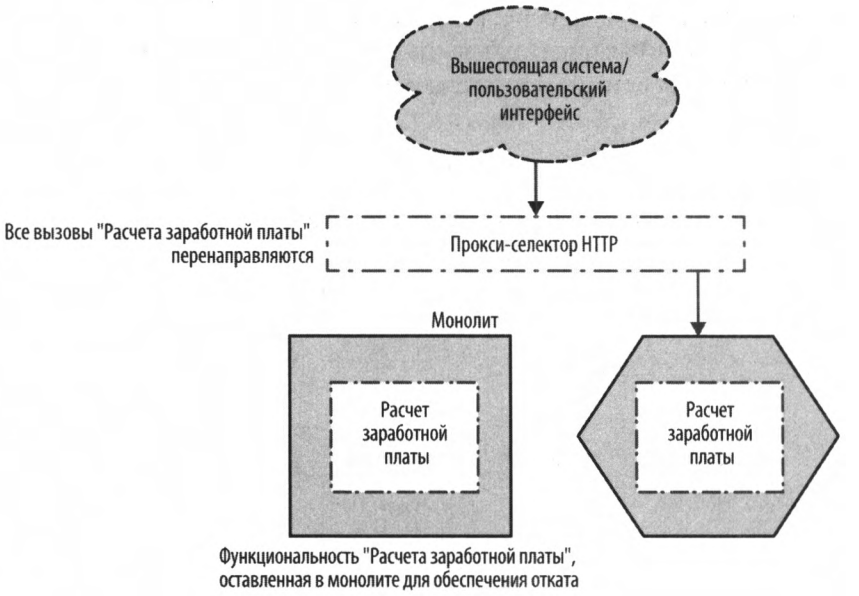


Рис. 3.8. Шаг 3: перенаправление вызова функциональности "Расчета заработной платы" в завершение миграции

титься к данным, которые в настоящее время хранятся в базе данных монолита? В *главе 4* мы разведаем варианты этого полнее.

Варианты прокси-селектора

Имплементация прокси-селектора частично будет зависеть от протокола, используемого монолитом. Если существующий монолит применяет HTTP, то у нас великолепное начало. HTTP — это такой широко поддерживаемый протокол, что у вас есть масса вариантов управления перенаправлением. Я бы, вероятно, выбрал выделенный прокси-селектор, например NGINX, который был создан с учетом именно таких вариантов использования и поддерживает целый ряд механизмов перенаправления, которые опробованы и протестированы и, вероятно, будут работать довольно хорошо.

Некоторые перенаправления будут проще, чем другие. Возьмем перенаправление URI-путей, возможно, как оно демонстрируется с использованием ресурсов REST. На рис. 3.9 мы переносим весь ресурс "Выписка счетов-фактур" в нашу новую службу, и его легко извлечь из URI-пути.

Если, однако, существующая система "закапывает" информацию о природе вызываемой функциональности где-то в теле запроса (возможно, в параметре формы), то наше правило перенаправления должно будет иметь возможность активировать параметр в методе POST, что сложнее имплементировать. Конечно, стоит проверить имеющиеся у вас варианты прокси-селекторов, чтобы убедиться, что они в состоянии справиться с такой ситуацией, если вы в ней окажетесь.

Если природа перехвата и перенаправления сложнее, или же в ситуациях, когда монолит использует не так хорошо поддерживаемый протокол, то вы, возможно, захотите закодировать что-то самостоятельно, но с этим подходом нужно быть очень осторожным. Ранее я написал несколько сетевых прокси-селекторов вручную (один на Java, другой на Python), и хотя это больше говорит о моей способности к кодированию, чем о чем-то еще, в обеих ситуациях прокси-селекторы были невероятно неэффективными, добавляя в систему значительный лаг. Если бы сегодня мне требовалось больше поведения, настроенного под свои нужды, то я скорее бы подумал о добавлении настроенной под свои нужды функциональности в выделенный прокси-селектор — например, NGINX позволяет использовать код, написанный на Lua, для добавления такого поведения.

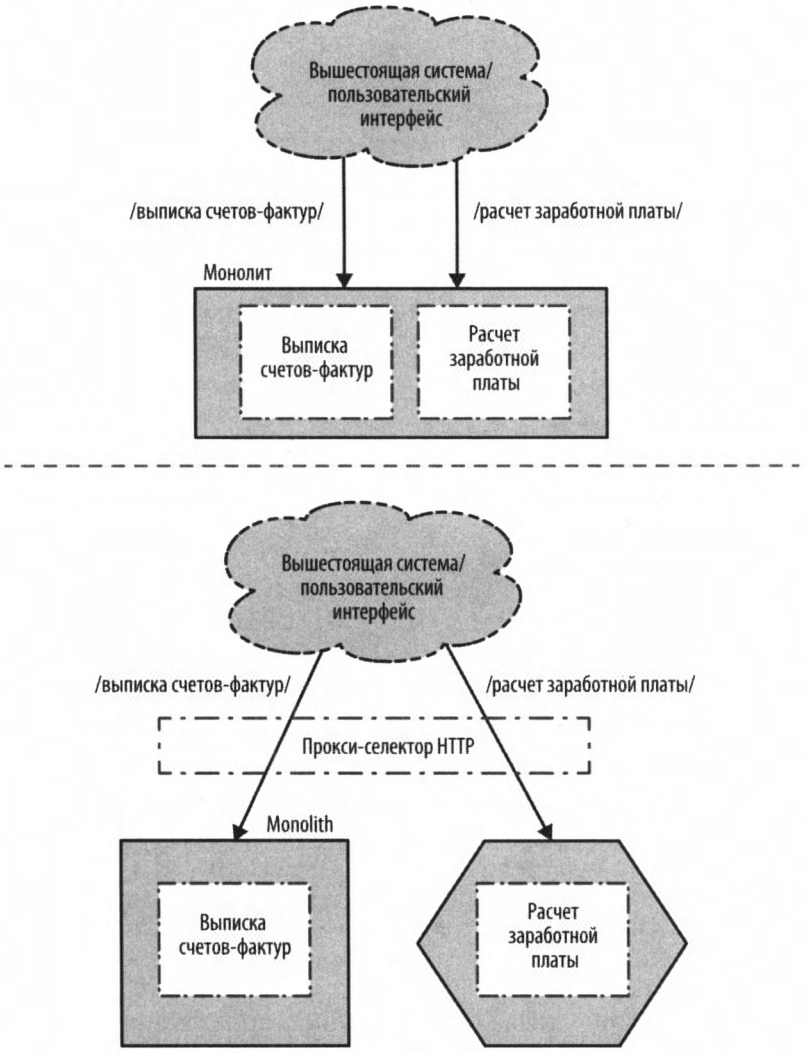


Рис. 3.9. Перенаправление в зависимости от ресурсов

Поступательное внедрение

Как видно из рис. 3.10, этот метод позволяет вносить архитектурные изменения с помощью серии малых шагов, каждый из которых предпринимается наряду с другими работами над системой.

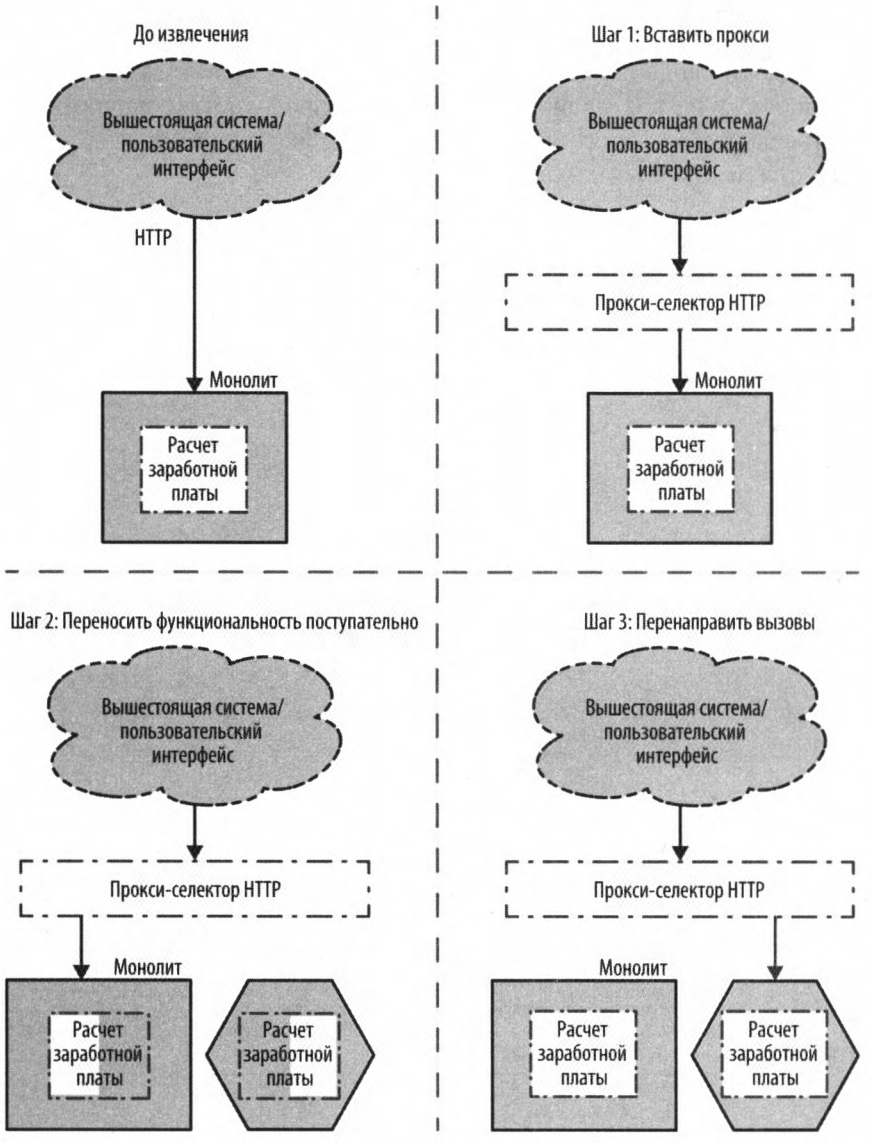


Рис. 3.10. Пошаговая имплементация "удавки" на основе HTTP

Вы, возможно, посчитаете, что новая имплементация функциональности "Расчета заработной платы", на которую вы переключились, по-прежнему остается слишком большой, и решите брать куски функциональности поменьше. Например, вы подумаете о мигрировании только части функционала "Расчета заработной платы" и пе-

реадресации вызовов соответствующим образом, имея часть поведения, имплементированным в монолите, и часть — в микрослужбе, как показано на рис. 3.11. Это вызовет затруднения, если функциональность, как в монолите, так и в микрослужбе, должна "видеть" один и тот же набор данных, поскольку для этого, вероятно, потребуется совместная база данных и все проблемы, которые она может принести.

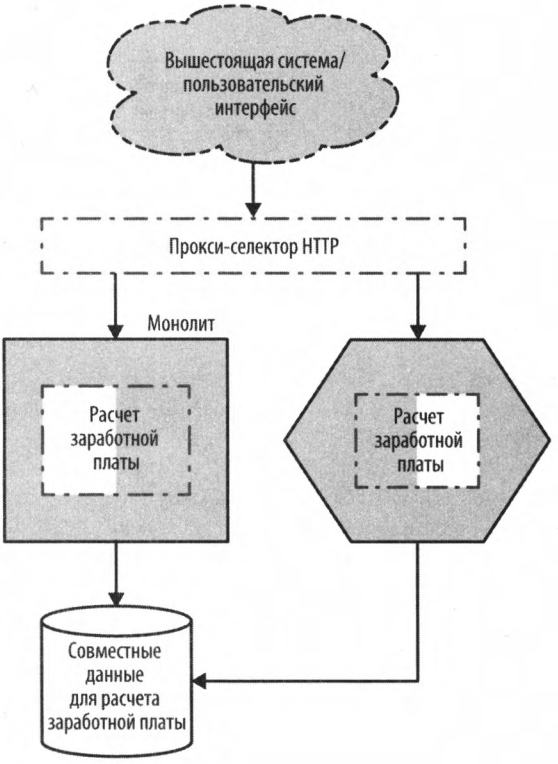


Рис. 3.11. Общий вид имплементации "удавки" на основе HTTP

Не требуется никакого переплатформирования[3] методом "Большого взрыва" с перекрытием. Это намного упрощает разбивку работы на этапы, которые будут поставляться в производство наряду с другими работами по доставке. Вместо того чтобы разбирать свои накопившиеся дела на "функциональные" и "технические", сложите всю эту работу вместе. Добейтесь успеха во внесении поступательных изменений в свою архитектуру, при этом по-прежнему поставляя новые функции!

Смена протоколов

Прокси-селектор также используется для трансформирования протокола. Например, вы, возможно, в настоящее время выставляете наружу SOAP-ориентированный HTTP-интерфейс, но вместо него ваша новая микрослужба собирается под-

[3] Переплатформирование (re-platforming) — это процесс, с помощью которого онлайновый бренд переходит с одной платформы на другую — *Пер.*

держивать gRPC-интерфейс. Тогда вы могли бы соответствующим образом сконфигурировать прокси-селектор для трансформирования запросов и откликов, как показано на рис. 3.12.

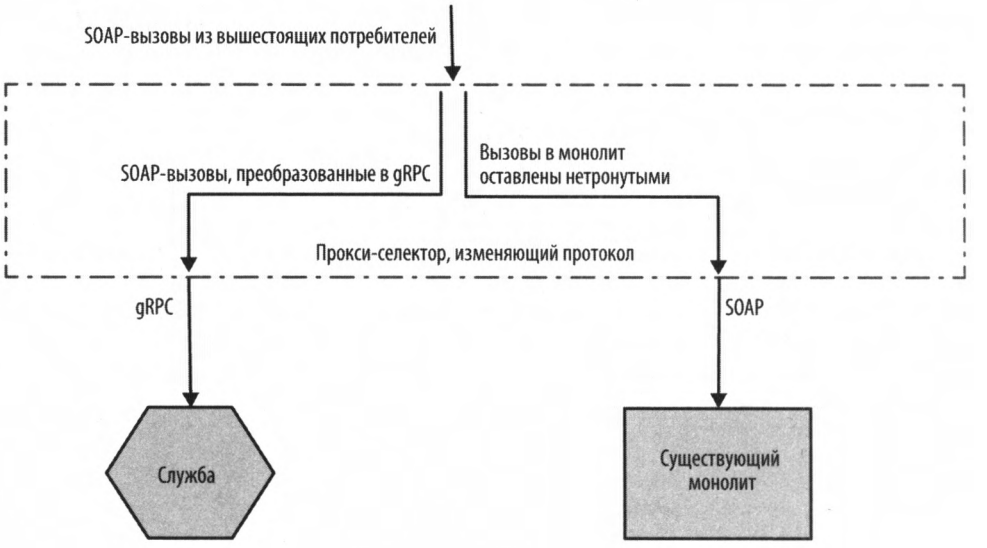


Рис. 3.12. Использование прокси-селектора для смены коммуникационного протокола в рамках миграции на основе "удавки"

У меня есть опасения по поводу этого подхода, в первую очередь из-за сложности и логики, которая начинает нарастать в самом прокси-селекторе. Для одной службы это выглядит не так уж и плохо, но если вы начнете трансформировать протокол для многочисленных служб, то работа, выполняемая в прокси-селекторе, будет нарастать и нарастать. Как правило, мы выполняем оптимизацию в сторону независимого развертывания наших служб, но если у нас совместный прокси-слой, который должен редактироваться несколькими группами, то он замедлит процесс внесения и развертывания изменений. Мы должны быть осторожными в том, чтобы не добавлять новый источник конкуренции. Когда мы обсуждаем архитектуру на основе микрослужб, часто звучит мантра "держите каналы безмозглыми, а конечные точки умными". Мы хотим уменьшить объем функциональности, заталкиваемый в совместные слои промежуточного софта, т. к. это по-настоящему замедлит разработку функций.

Если вы хотите мигрировать используемый протокол, то я бы предпочел вложить это отображение в саму службу — при этом служба будет поддерживать как ваш старый коммуникационный протокол, так и новый. Внутри службы вызовы нашего старого протокола будут внутренне переотображаться в новый коммуникационный протокол, как мы видим на рис. 3.13. Это позволяет избежать необходимости управлять изменениями в прокси-слоях, используемых другими службами, и предоставляет службе полный контроль над тем, как эта функциональность изменяется с течением времени. Микрослужбы можно рассматривать как коллекцию функциональности на конечной точке сети. Одну и ту же функциональность можно выстав-

лять наружу по-разному разным потребителям; поддерживая разные форматы сообщений или запросов внутри этой службы, мы, в сущности, просто поддерживаем разные потребности наших вышестоящих потребителей.

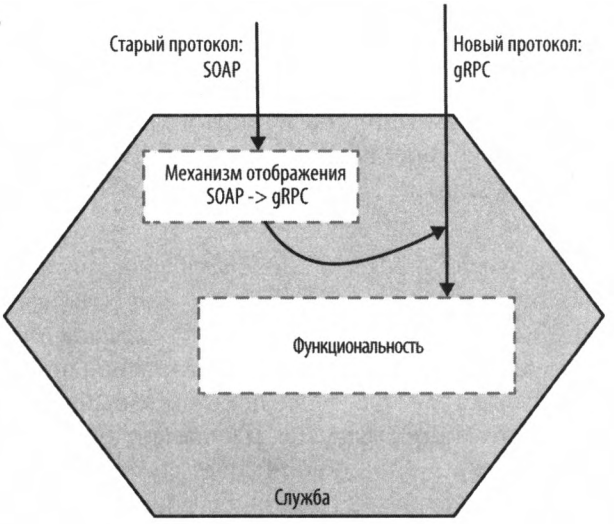


Рис. 3.13. Если вы хотите сменить тип протокола, то подумайте о выставлении службой своих возможностей наружу по нескольким типам протокола

Вкладывание специфичного для службы отображения запросов в отклики внутрь службы позволяет делать прокси-слой простым и гораздо более обобщенным. Вдобавок, в условиях, когда служба предоставляет оба типа конечных точек, вы даете себе время на то, чтобы мигрировать вышестоящих потребителей перед потенциальным выводом из эксплуатации старого API.

И сетки для служб

В компании Square был принят гибридный подход к решению этой проблемы[4]. Они решили отказаться от своего собственного доморощенного RPC-механизма двухсторонней коммуникации между службами в пользу принятия gRPC, хорошо поддерживаемого каркаса RPC с открытым исходным кодом с очень широкой экосистемой. Для того чтобы сделать это как можно безболезненнее, они хотели уменьшить число изменений, необходимых в каждой службе. Для этого они использовали сетку для служб[5].

С помощью сетки для служб, показанной на рис. 3.14, каждый экземпляр службы осуществляет коммуникацию с другими экземплярами службы через свой собст-

[4] Более подробное объяснение дано в статье Сноу Петтерсена (Snow Pettersen) "Дорога к сетке для служб в Envoy" (The Road to an Envoy Service Mesh, **https://squ.re/2nts1Gc**) в блоге для разработчиков на веб-сайте Square.

[5] Сетка для служб (service mesh) — это выделенный инфраструктурный слой для облегчения сквозного обмена данными между микрослужбами, часто использующий прицепной (sidecar) прокси-селектор — *Пер.*

венный выделенный локальный прокси-селектор. Каждый экземпляр прокси-селектора можно сконфигурировать специально для экземпляра службы, с которым он сотрудничает. Вы также можете обеспечить централизованное управление и мониторинг этих прокси-селекторов с помощью плоскости управления. Поскольку центрального прокси-слоя нет, вы избегаете ловушек, связанных с наличием совместного "умного" канала. Если это необходимо, то каждая служба практически способна владеть своей собственной частью канала службы. Стоит отметить, что по мере развития архитектуры компания Square в итоге вынуждена была создать свою собственную сетку для служб, специфичную для ее потребностей, хотя и с прокси-селектором с открытым исходным кодом Envoy вместо устоявшегося решения, такого, как Linkerd или Istio.

Популярность сетки для служб растет, и концептуально, как мне думается, эта идея попала в точку. Это отличный способ решения общих вопросов двухсторонней коммуникации между службами. Беспокоит то, что, несмотря на большую работу ряда очень умных людей, потребовалось продолжительное время для стабилизации инструментария в этом пространстве. Istio, похоже, явный лидер, но он далеко не единственный вариант в этом пространстве, и существуют новые инструменты, которые появляются еженедельно. Мой общий совет состоит в том, чтобы, если можно, дать идее сетки для служб немного больше времени на то, чтобы устояться, прежде чем сделать свой выбор.

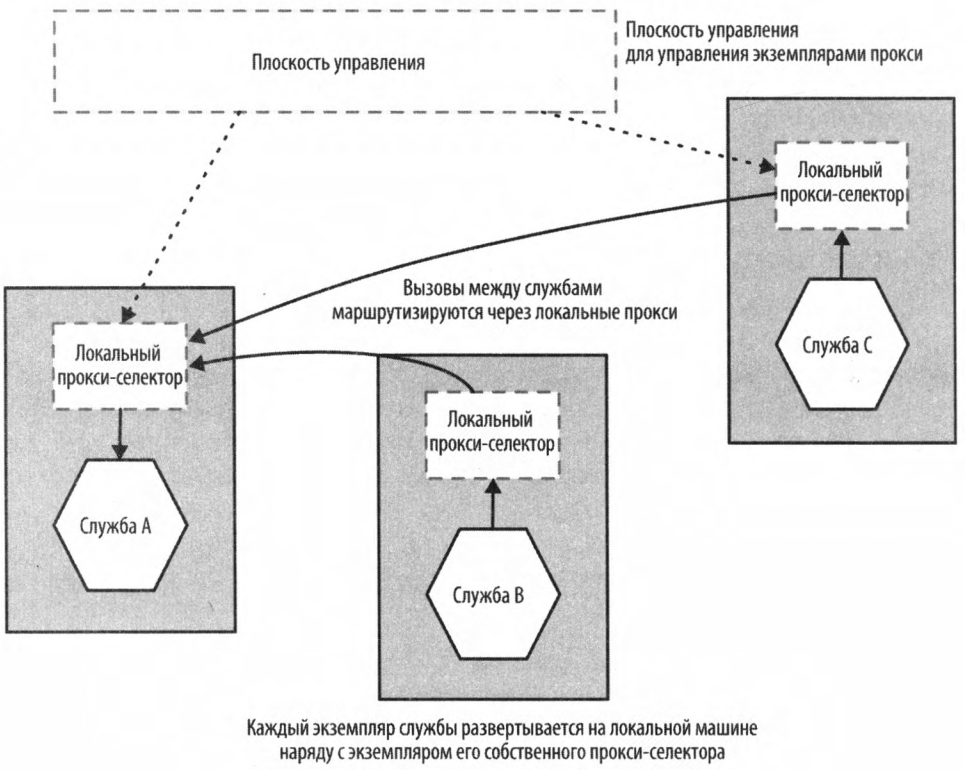


Рис. 3.14. Общий вид сетки для служб

Пример: FTP

Хотя я уже подробно рассказывал об использовании шаблона "Удавка" для систем на основе HTTP, вам ничто не мешает перехватывать и перенаправлять другие формы коммуникационных протоколов. Швейцарская риэлторская компания Homegate с помощью вариации этого шаблона изменила способ закачки клиентами новых листингов недвижимости.

Клиенты Homegate закачивали листинги по FTP, при этом существующая монолитная система занималась закачанными файлами. Указанная компания стремилась перейти на микрослужбы, а также хотела начать поддерживать новый механизм закачки, который вместо пакетной закачки по FTP собирался использовать REST API, соответствующий стандарту, ратификация которого ожидалась в ближайшем будущем.

Риэлторская компания не собиралась ничего менять с точки зрения клиента — она хотела сделать любые изменения бесстыковыми. Это означает, что механизм FTP по-прежнему оставался востребованным; с помощью него клиенты взаимодействовали с системой, по крайней мере на текущий момент. В конце компания перехватывала FTP-закачки (обнаруживая изменения в журнале FTP-сервера) и направляла только что закачанные файлы в адаптер, который преобразовывал закачанные файлы в запросы к новому REST API, как показано на рис. 3.15.

Рис. 3.15. Перехватывание FTP-закачки и переадресация ее на службу новых листингов для Homegate

С точки зрения клиента, сам процесс закачки не изменился. Выгода заключалась в том, что новая служба, которая занималась клиентской закачкой, публиковала обновленные данные существенно быстрее, помогая клиентам выводить их рекламу в прямой эфир намного скорее. В дальнейшем планировалось выставить наружу новый REST API клиентам напрямую. Интересно, что в этот период задействовались оба механизма закачки листингов. Это позволило группе обеспечить условие, чтобы оба механизма закачки работали надлежащим образом. Указанный пример — отличная демонстрация шаблона, который мы рассмотрим далее в разделе "Шаблон: параллельное выполнение".

Пример: перехват сообщений

До сих пор мы рассматривали перехват синхронных вызовов, но что если ваш монолит управляется из какой-то другой формы протокола, возможно, получая сообщения через брокера сообщений? Фундаментальный шаблон тот же — нам нужен метод перехвата вызовов и перенаправления их в нашу новую микрослужбу. Главное различие заключается в природе самого протокола.

Маршрутизация на основе содержимого

На рис. 3.16 наш монолит получает многочисленные сообщения, подмножество которых нам необходимо перехватить.

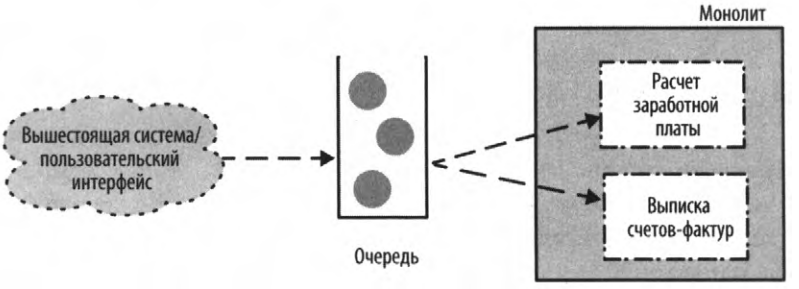


Рис. 3.16. Монолит, принимающий вызовы через очередь

Простой подход состоит в перехвате всех сообщений, предназначенных для нижестоящего монолита, и фильтрации сообщений в надлежащее место, как показано на рис. 3.17. Это, в сущности, имплементация шаблона маршрутизатора, основанного на содержимом (content-based router), как описано в книге Бобби Вульфа и Грегора Хопа "Шаблоны интеграции предприятия"[6].

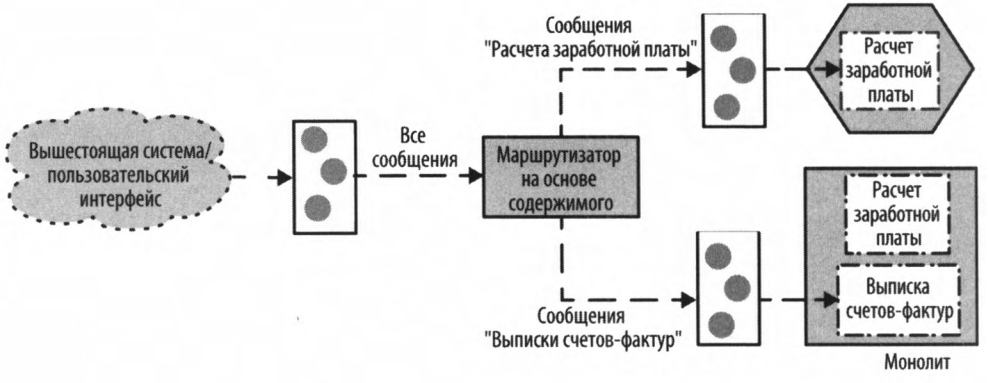


Рис. 3.17. Использование маршрутизатора, основанного на содержимом, для перехвата вызовов, связанных с обменом сообщениями

[6] См. "Шаблоны интеграции предприятия" (Enterprise Integration Patterns, Bobby Woolf, Gregor Hohpe, Addison-Wesley, 2003).

Такой метод позволяет оставить монолит нетронутым, но мы помещаем на наш путь запросов еще одну очередь, что вносит добавочную задержку, и это еще одна вещь, которой нам нужно управлять. Другая озабоченность заключается в том, сколько "мозгов" мы помещаем в слой обмена сообщениями. В *главе 4* книги "Создание микросервисов" я говорил о трудностях, связанных с использованием системой излишних "мозгов" в сетях между вашими службами, поскольку это затруднит понимание систем и затруднит их изменение.

Вместо этого я призвал вас принять мантру "умные конечные точки, безмозглые каналы", на чем я по-прежнему настаиваю. Здесь можно поспорить, что маршрутизатор на основе содержимого означает, что мы имплементируем "умный канал", добавляя сложность с точки зрения того, как вызовы маршрутизируются между нашими системами. В некоторых ситуациях этот метод очень полезен, но решать вам, где находится золотая середина.

Селективное потребление

Альтернативой было бы изменить монолит и побудить его игнорировать отправляемые сообщения, которые должны быть получены нашей новой службой, как мы видим на рис. 3.18. Здесь и наша новая служба, и наш монолит делят между собой одну и ту же очередь, и локально они используют процесс своего рода сопоставления с шаблоном для прослушивания сообщений, в которых они заинтересованы. Этот вид фильтрации является довольно распространенным требованием в системах на основе сообщений и имплементируется с использованием чего-то вроде "Селектора сообщений" в JMS или эквивалентной технологии на других платформах.

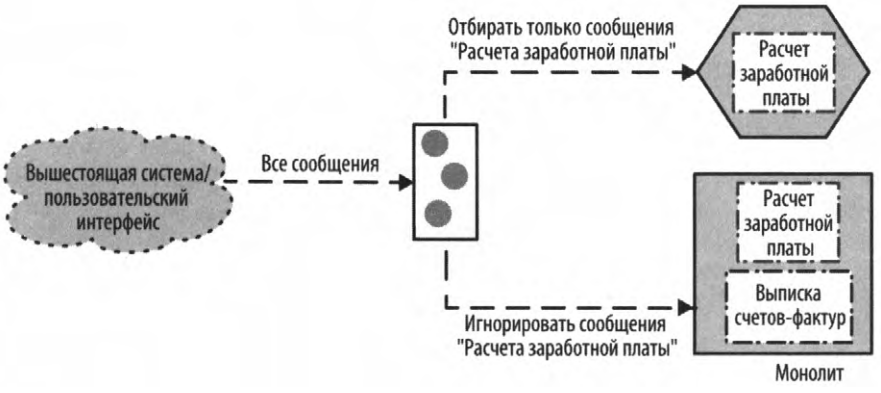


Рис. 3.18. Вариант использования маршрутизатора, основанного на содержимом, для перехвата вызовов, связанных с обменом сообщениями

Такой подход к фильтрации снижает необходимость создания дополнительной очереди, но имеет ряд трудностей. Во-первых, ваша опорная технология обмена сообщениями может и не позволить вам совместно использовать одну единственную подписку на очередь такого рода (хотя эта функция является общепринятой, поэтому я был бы удивлен, если бы это было так). Во-вторых, когда вы хотите перенаправлять вызовы, это требует, чтобы два изменения были достаточно хорошо ско-

ординированы. Вам нужно остановить чтение монолитом вызовов, предназначенных для новой службы, а затем побудить службу их забрать. Схожим образом, для отката перехвата вызовов требуется откат двух изменений.

Чем больше типов потребителей у вас для одной и той же очереди и чем сложнее правила фильтрации, тем проблематичнее все становится. Можно легко представить себе ситуацию, в которой два потребителя начинают получать одно и то же сообщение из-за перекрывающих друг друга правил или даже наоборот — некоторые сообщения игнорируются полностью. По этой причине я бы, вероятно, подумал об использовании селективного потребления только с малым числом потребителей и/или с простым набором правил фильтрации. Подход с маршрутизацией на основе содержимого, вероятно, будет иметь больше смысла по мере увеличения числа типов потребителей, хотя остерегайтесь упомянутых ранее потенциальных недостатков, в особенности относимых к проблеме "умных каналов".

Добавочная сложность с этим решением или маршрутизацией на основе содержимого заключается в том, что если мы используем асинхронный коммуникационный стиль запросов-откликов, то нам нужно обеспечить, чтобы мы направляли запрос обратно клиенту, как мы надеемся, без осознания им факта изменения. Существуют и другие варианты маршрутизации вызовов в системах, обусловленных сообщениями, системах, многие из которых помогают вам имплементировать миграции на основе шаблона "Фикус-удавка". Здесь я горячо рекомендую книгу "Шаблоны интеграции предприятия" в качестве великолепного ресурса.

Другие протоколы

Как я надеюсь, вы поняли из рассмотренного примера, есть много способов перехвата вызовов в существующем монолите, даже если вы используете разные типы протоколов. Что делать, если ваш монолит управляется пакетной закачкой файлов? Перехватить пакетный файл, извлечь вызовы, которые вы хотите перехватить, и удалить их из файла перед его пересылкой дальше. Правда, некоторые механизмы еще более усложняют этот процесс, и будет намного проще, если использовать что-то вроде HTTP, но при некотором творческом подходе шаблон "Фикус-удавка" может применяться в очень многих ситуациях.

Другие примеры шаблона "Фикус-удавка"

Шаблон "Фикус-удавка" очень полезен в любом контексте, где вы хотите поступательно переплатформировать существующую систему, и его применение не ограничивается группами, имплементирующими архитектуры на основе микрослужб. Указанный шаблон использовался уже в течение длительного времени, прежде чем Мартин Фаулер описал его в 2004 году. У моего предыдущего работодателя, ThoughtWorks, мы часто с помощью данного шаблона перестраивали монолитные приложения. Пол Хаммант (Paul Hammant) — автор неисчерпаемого списка проектов[7], где мы использовали этот шаблон в его блоге. Проекты включают в себя ор-

[7] См. **http://bit.ly/2paBpyP**.

дерную книгу трейдинговой компании, приложение для бронирования авиабилетов, систему покупки железнодорожных билетов и портал объявлений.

Изменение поведения во время мигрирования функциональности

Здесь и в других местах книги я сосредоточусь на шаблонах, которые я выбрал именно в силу того, что они используются, чтобы осуществлять поступательную миграцию существующей системы на архитектуру, основанную на микрослужбах. Одной из главных причин является возможность смешивать миграционную работу с продолжающейся доставкой функций. Но все же есть одна проблема, которая возникает, когда вы хотите изменить или обогатить поведение активно мигрируемой системы.

Представьте себе, например, что мы собираемся использовать шаблон "Фикус-удавка", для того чтобы перенести нашу существующую функциональность "Расчета заработной платы" из монолита. Шаблон "Фикус-удавка" дает нам возможность делать это в несколько шагов, причем каждый шаг теоретически позволяет нам откатиться назад. Если бы мы развернули новую службу "Расчет заработной платы" для наших клиентов и обнаружили с ней проблему, то мы могли бы перенаправить вызовы функциональности "Расчета заработной платы" обратно в старую систему. Это хорошо работает, если функциональность "Расчета заработной платы" у монолита и микрослужбы является функционально эквивалентной, но что если мы изменили поведение "Расчета заработной платы" в рамках миграции?

Если в микрослужбе "Расчет заработной платы" было устранено несколько дефектов, причем их устранение коснулось того, как она работает, и эти изменения не были перенесены назад в эквивалентную функциональность в монолите, то откат снова привел бы к появлению этих дефектов в системе. Все было бы еще проблематичнее, если в микрослужбу "Расчет заработной платы" вы добавили новую функциональность — откат тогда потребовал бы удаления функций из ваших клиентов.

Здесь нет простого решения. Если вы допускаете изменения в функциональности, которую вы переносите до завершения миграции, то должны смириться с тем, что затрудняете любой откат. Проще, если вы не допускаете никаких изменений до завершения миграции. Чем дольше длится миграция, тем сложнее обеспечить "заморозку функции" в этой части системы — если есть спрос на изменение части вашей системы, то этот спрос вряд ли исчезнет. Чем дольше вы будете завершать миграцию, тем больше давления вы будете испытывать в том, чтобы "вставить эту функцию, пока мы тут". Чем меньше длится каждая ваша миграция, тем меньше давление, которое вы будете испытывать в том, чтобы изменить функциональность, переносимую до завершения миграции.

Во время мигрирования функциональности старайтесь исключать любые изменения в переносимом поведении. Если возможно, откладывайте новые функции или устранение дефектов до завершения миграции. В противном случае вы уменьшите свою способность откатывать изменения назад в вашей системе.

Шаблон: "Композиция пользовательского интерфейса"

С помощью методов, рассматриваемых до сих пор, мы в первую очередь "заталкивали" работу по поступательной миграции на сервер, однако пользовательский интерфейс предоставляет нам некоторые полезные возможности для склеивания функциональности, раздаваемой частично из существующего монолита или новой архитектуры, основанной на микрослужбах.

Много лет назад я принимал участие в помощи, оказываемой газете Guardian в онлайновом переходе с существующей системы управления контентом на новую заказную платформу на базе Java. Это должно было совпасть с развертыванием совершенно нового внешнего вида и ощущения от онлайновой газеты в связке с перезапуском печатного издания. Поскольку мы хотели принять поступательный подход, пробежка от существующей CMS к новому веб-сайту раздаваемому совершенно по-новому, была фазирована по частям, ориентируясь на конкретные тематические вертикали (путешествия, новости, культура и т. д.). Даже внутри этих вертикалей мы также искали возможности разложить миграцию на меньшие "куски".

В итоге мы воспользовались двумя полезными композиционными методами. Из разговоров с другими компаниями мне за последние несколько лет стало ясно, что вариации на тему этих методов в значительной мере служат индикатором того, сколько организаций используют архитектуры на основе микрослужб.

Пример: страничная композиция

Хотя в случае с Guardian мы начали с развертывания отдельного виджета (что мы вскоре обсудим), план всегда состоял в том, чтобы для вывода в прямой эфир совершенно нового внешнего вида и ощущения использовать главным образом одностраничную миграцию. Это было сделано на вертикальной основе, причем вертикаль "Путешествия" была первой, которую мы отправили в "прямой эфир". Посетителям веб-сайта в течение этого транзитного периода показывались другие варианты внешнего вида и ощущения, когда они заходили на новые части веб-сайта. Кроме того, были предприняты большие усилия, для того чтобы все старые ссылки на страницы были перенаправлены на новые места (там, где URL-адреса поменялись).

Когда Guardian внесла еще одно изменение в технологии, несколько лет спустя отойдя от монолита на базе Java, они снова воспользовались аналогичным методом миграции по одной вертикали за один раз. На этом этапе для имплементации новых правил маршрутизации они использовали сеть быстрой доставки контента (CDN), эффективно задействуя CDN так же, как вы бы применяли межкорпоративный прокси-селектор[8].

[8] Было приятно услышать от Грэма Тэкли (Graham Tackley) из Guardian, что "новая" система, помощь в имплементации которой я оказывал с самого начала, действовала почти 10 лет, прежде чем ее полностью заменили на текущую архитектуру. Как читатель их веб-сайта, я поймал себя на мысли, что за этот период я ни разу не заметил никаких изменений!

В австралийской компании REA Group, которая служит провайдером онлайновых листингов недвижимости, разные группы разработчиков отвечают за листинги коммерческой или жилой недвижимости и владеют этими каналами целиком. В такой ситуации имеет смысл подход на основе странично-ориентированной композиции, поскольку группа владеет всем сквозным опытом. REA фактически задействует тонко различающийся брендинг для разных каналов, а это означает, что странично-ориентированная декомпозиция имеет еще больший смысл, поскольку разным группам клиентов можно поставлять совершенно разный опыт.

Пример: виджетная композиция

В Guardian вертикаль "Путешествия" была первой, которая была выявлена для миграции на новую платформу. Отчасти это объяснялось тем, что у нее были некоторые трудности с разбиением на категории, но также и тем, что эта часть веб-сайта была не самой "громкой". В принципе, мы хотели вывести хоть что-то в "прямой эфир", научиться на этом опыте, а также обеспечить бесперебойную работу праймовых частей веб-сайта, если что-то пойдет не так.

Вместо того чтобы выводить в "прямой эфир" всю туристическую часть веб-сайта с изобилием подробных репортажей о гламурных направлениях по всему миру, мы хотели сделать гораздо более сдержанный релиз с целью тестирования системы. Вместо этого мы развернули отдельный виджет, показывающий 10 лучших туристических направлений, определяемых с помощью новой системы. Этот виджет был вклеен в старые страницы газеты о путешествиях, как показано на рис. 3.19. В нашем случае мы использовали метод, именуемый боковыми вставками (Edge-Side Includes, ESI), используя Apache. С помощью ESI вы определяете на своей веб-странице шаблон, и веб-сервер "вклеивает" этот контент.

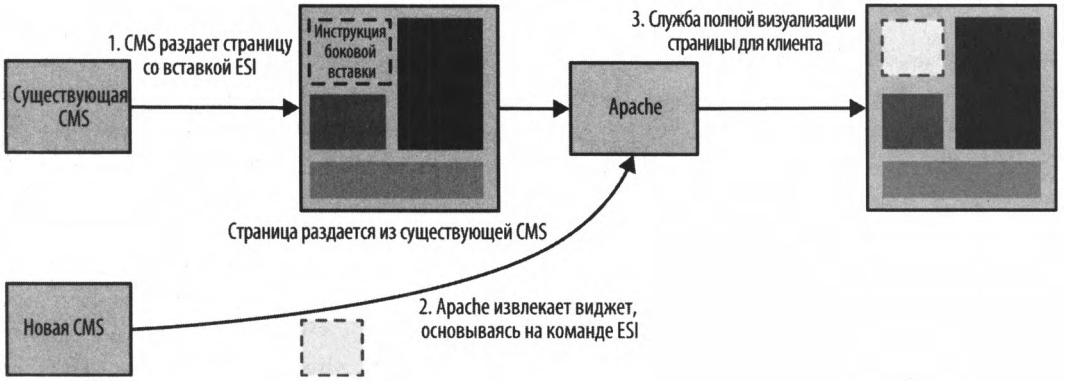


Рис. 3.19. Использование боковых вставок ESI для вклеивания контента из новой CMS газеты Guardian

В настоящее время вклеивание виджета исключительно на стороне сервера, похоже, встречается реже. Это в значительной степени связано с тем, что браузерная технология стала более изощренной, позволяя делать гораздо больше композиции в самом браузере (или в нативном приложении — подробнее об этом позже). Это

означает, что для виджетно-ориентированных веб-интерфейсов сам браузер нередко делает многочисленные вызовы для загрузки различных виджетов посредством целого ряда методов. Виджетная композиция имеет еще одну выгоду: если один виджет не загружается — возможно, потому, что поддерживающая служба недоступна, — другие виджеты по-прежнему отрисовываются, допуская только частичную, а не полную деградацию службы.

Хотя в Guardian мы, в конце концов, использовали главным образом страничноориентированную композицию, многие другие компании активно применяют виджетно-ориентированную композицию с поддержкой бэкэндовых служб. Orbitz (теперь часть Expedia), например, создала выделенные службы для раздачи лишь одного виджета[9]. До перехода на микрослужбы веб-сайт Orbitz уже был разбит на отдельные "модули" UI (в номенклатуре Orbitz). Эти модули UI представляют форму поиска, форму бронирования, карту и т. д. Они первоначально раздавались непосредственно из службы оркестровки контента, как показано на рис. 3.20.

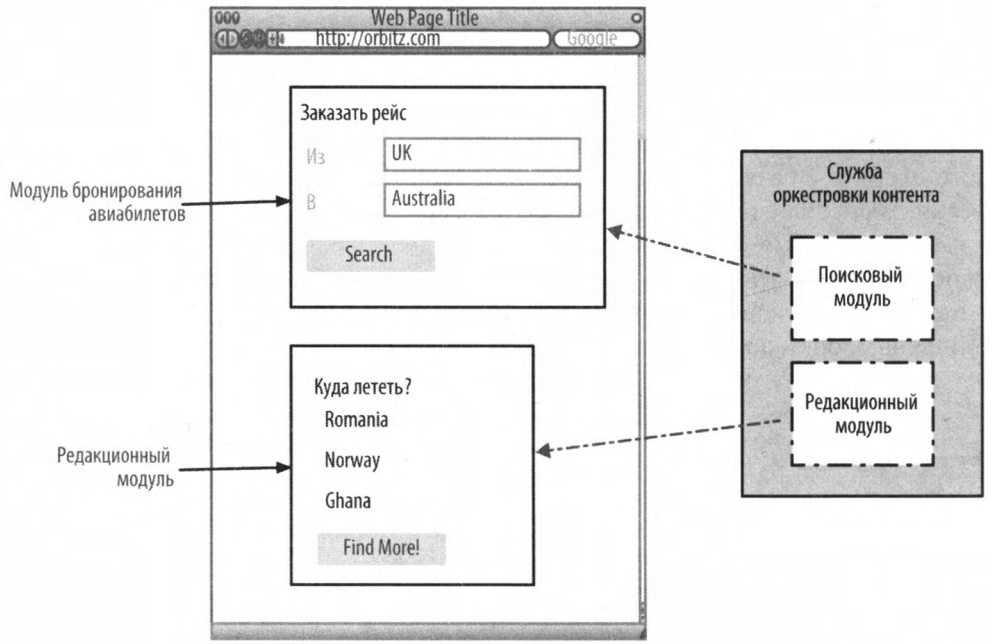


Рис. 3.20. До миграции на микрослужбы служба оркестровки контента Orbitz раздавала все модули

Служба "Оркестровка контента" фактически была крупным монолитом. Группы, которые владели этими модулями, должны были координировать изменения, вносимые внутри монолита, что приводило к значительным задержкам во внедрении изменений. Это классический пример проблемы конкуренции за доставку, которую я высветил в *главе 1*: всякий раз, когда группам приходится координировать вне-

[9] См. Стив Хоффман (Steve Hoffman) и Рик Фаст (Rick Fast), "Задействование микрослужб в Orbitz" (Enabling Microservices at Orbitz, **http://bit.ly/2nGNgnI**), YouTube, 11 августа 2016.

дрение изменений, стоимость изменений растет. В рамках стремления к более быстрым релизным циклам, когда Orbitz решила попробовать микрослужбы, они сосредоточили свою декомпозицию на этих модулях, начиная с редакционного модуля. Служба "Оркестровка контента" была изменена, делегируя ответственность за транзитные модули нижестоящим службам, как мы видим на рис. 3.21.

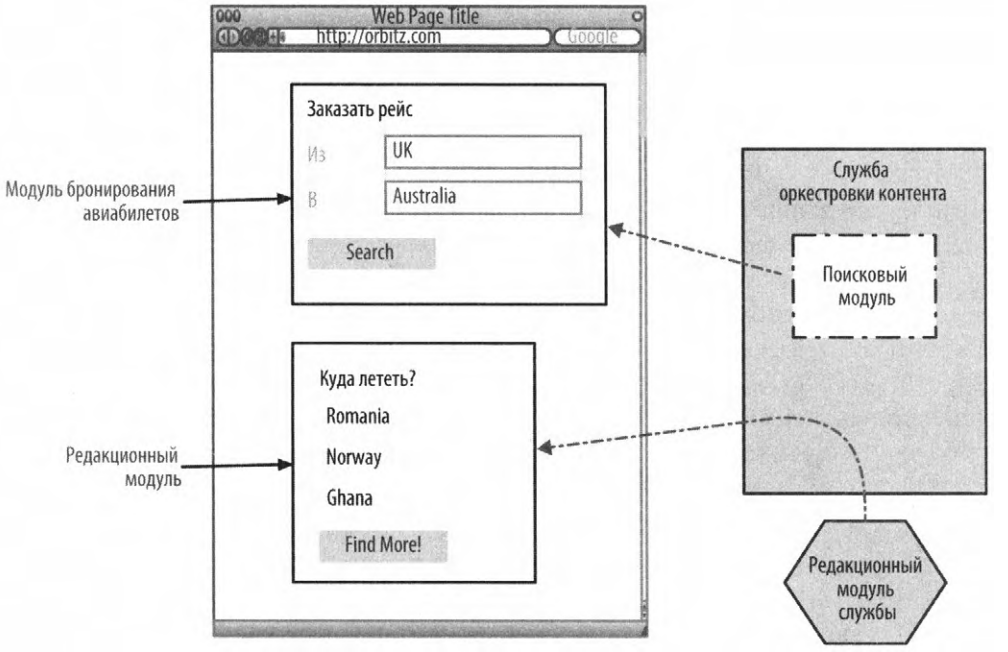


Рис. 3.21. Модули мигрировались по одному, при этом служба оркестровки контента делегировалась новым поддерживающим службам

Тот факт, что UI уже был разложен визуально по этим линиям, упростил данную работу в поступательном режиме. Транзит был облегчен еще больше, поскольку отдельные модули уже имели чистые линии владения, что облегчало миграцию, не мешая другим группам.

Стоит отметить, что не все пользовательские интерфейсы подходят для декомпозиции в чистые виджеты, но, когда они для этого подходят, это значительно облегчает работу поступательной миграции на микрослужбы.

И мобильные приложения

Хотя мы главным образом говорили о веб-интерфейсе пользователя, некоторые из этих методов могут хорошо работать и для мобильных клиентов. Например, как Android, так и iOS предоставляют возможность компонентизации частей своих пользовательских интерфейсов, облегчая работу над этими частями пользовательского интерфейса в изоляции или рекомбинированными различными способами.

Одна из трудностей развертывания изменений с помощью нативных мобильных приложений заключается в том, что как магазин Apple App Store, так и магазин

Google Play требуют, чтобы приложения были предъявлены на рассмотрение и подтверждены до появления новых версий. Хотя за последние несколько лет время, в течение которого магазины приложений подписывают соглашения, в целом значительно сократилось, это все же по-прежнему увеличивает время, прежде чем новые релизы программно-информационного обеспечения могут быть развернуты.

Само приложение в этой точке также является монолитом: если вы хотите изменить одну единственную часть собственного мобильного приложения, то все приложение по-прежнему должно быть развернуто целиком. Вам также приходится учитывать тот факт, что для появления новых функций пользователям приходится скачивать новое приложение. Такая ситуация не возникает при использовании веб-приложения, т. к. изменения органично доставляются в браузер пользователей.

Многие организации решали эту проблему, позволяя вносить динамические изменения в существующее нативное мобильное приложение, не прибегая к развертыванию новой версии нативного приложения. При развертывании изменений на стороне сервера клиентские устройства сразу же видят новую функциональность без необходимости развертывания новой версии нативного мобильного приложения. Это достигается просто с помощью таких вещей, как компоненты WebView, позволяющие встраивать веб-страницы в приложения, хотя некоторые компании используют более изощренные методы.

Пользовательский интерфейс Spotify на всех платформах сильно ориентирован на компоненты, включая ее приложения для iOS и Android. Практически все, что вы видите, — это отдельный компонент, от простого текстового заголовка до обложки альбома или списка воспроизведения[10]. Некоторые из этих модулей, в свою очередь, поддерживаются одной или несколькими микрослужбами. Конфигурация и расположение этих компонентов UI определяются декларативно на стороне сервера; инженеры Spotify способны быстро изменять изображения, которые видят пользователи, и откатывать их, не требуя отправки новых версий своего приложения в магазин приложений. Это позволяет им гораздо быстрее экспериментировать и апробировать новые функции.

Пример: микрофронтэнды

По мере того как улучшались пропускная способность и возможности веб-браузеров, совершенствовалась и изощренность кода, выполняемого в браузерах. Многие веб-интерфейсы пользователей в настоящее время используют некоторую форму каркаса одностраничного приложения, что устраняет концепцию приложения, состоящего из разных веб-страниц. Вместо этого у вас более мощный пользовательский интерфейс, где все работает в одной панели — практически внутрибраузерный пользовательский опыт, который ранее был доступен только для тех из нас,

[10] См. статью Джона Санделла (John Sundell) "Строительство компонентно-обусловленных UI в Spotify" (Building Component-Driven UIs at Spotify, **http://bit.ly/2nDpJUP**), опубликованную 25 августа 2016 года.

кто работал с "толстыми" SDK пользовательского интерфейса, такими, как Swing на Java.

Поставляя весь интерфейс на одной странице, мы, очевидно, не можем рассматривать страничную композицию, поэтому нам придется подумать о некой форме виджетной композиции. Были предприняты попытки кодифицировать общепринятые форматы виджетов для веб. В последнее время спецификация веб-компонентов пытается определить стандартную компонентную модель, поддерживаемую всеми браузерами. Однако потребовалось много времени на то, чтобы этот стандарт набрал обороты в условиях, когда браузерная поддержка (среди прочего) была значительным камнем преткновения.

Люди, пользующиеся каркасами одностраничных приложений, такими, как Vue, Angular или React, не сидят и не ждут, пока веб-компоненты решат их проблемы. Вместо этого многие из них пытались разобраться с вопросом, как модуляризировать пользовательские интерфейсы, построенные с помощью SDK-инструментариев, которые изначально были спланированы так, чтобы владеть всей браузерной панелью. Это привело к толчку в сторону того, что некоторые окрестили как микрофронтэнды.

На первый взгляд, микрофронтэнды на самом деле просто разбивают пользовательский интерфейс на разные компоненты, над которыми можно работать независимо. В этом нет ничего нового, ведь компонентно-ориентированный софт предшествует моему появлению на свет на несколько лет! Гораздо интереснее, что люди вырабатывают способы побудить веб-браузеры, SDK SPA и компонентизацию работать вместе. Как именно создать отдельный пользовательский интерфейс из кусков Vue и React, не сталкивая их зависимости друг с другом, но все же позволяя им потенциально обмениваться информацией?

Подробное освещение этой темы выходит за рамки данной книги, отчасти потому, что точный способ выполнения такой работы будет зависеть от используемых каркасов SPA. Но если вы обнаружите, что у вас есть одностраничное приложение, которое вы хотите разбить на куски, то вы не одиноки, и существует много людей, которые делятся методами и библиотеками, для того чтобы это работало.

Где его использовать

Композиция пользовательского интерфейса как метод, позволяющий переплатформировать системы, очень эффективна, поскольку позволяет переносить целые вертикальные куски функциональности. Однако, чтобы он работал как надо, необходима возможность изменять существующий пользовательский интерфейс для обеспечения безопасной вставки новой функциональности. Мы рассмотрим композиционные методы далее в этой книге, но стоит отметить, что используемые вами методы часто зависят от природы технологии, задействуемой для имплементации пользовательского интерфейса. Хороший старомодный веб-сайт упрощает композицию пользовательского интерфейса, в то время как технология одностраничных приложений добавляет некоторую сложность и часто препятствует различным подходам к имплементации!

Шаблон: "Ветвление по абстракции"

Для того чтобы полезный шаблон "Фикус-удавка" работал как надо, необходима способность перехватывать вызовы по периметру нашего монолита. Однако, что произойдет, если функциональность, которую мы хотим извлечь, находится глубже внутри нашей существующей системы? Возвращаясь к предыдущему примеру, возьмем желание извлечь функциональность "Уведомления", как показано на рис. 3.4.

Для того чтобы выполнить такое извлечение, нам нужно будет внести изменения в существующую систему. Эти изменения бывают значительными и препятствующими для других разработчиков, работающих над кодовой базой в одно и то же время. Здесь у нас возникает конкурирующая напряженность. С одной стороны, мы хотим вносить изменения поступательными шагами. С другой стороны, мы хотим уменьшить помехи для других людей, работающих в других областях кодовой базы. Это естественным образом приведет нас к желанию завершить работу быстро.

Часто во время переработки частей существующей кодовой базы люди будут делать эту работу на отдельной ветви исходного кода. Это позволяет вносить изменения, не нарушая работу других разработчиков. Проблема заключается в том, что как только изменение в какой-либо ветви завершено, эти изменения должны быть слиты обратно, что часто вызывает значительные трудности. Чем дольше существует ветвь, тем больше эти проблемы. Я не буду сейчас подробно останавливаться на проблемах, связанных с долгоживущими ветвями исходного кода, скажу лишь, что они противоречат принципам непрерывной интеграции. Данные, взятые из "Отчета о состоянии DevOps за 2017 год"[11], также показывают, что внедрение разработки на основе главной ветви исходного кода (где изменения вносятся непосредственно на магистральной линии и ветвление избегается) и использование короткоживущих ветвей способствует более высокой производительности ИТ-групп. Скажем так, я не поклонник долгоживущих ветвей, и я не одинок.

Таким образом, мы хотим иметь возможность вносить изменения в кодовую базу поступательным образом, но при этом держать на минимуме нарушение работы у разработчиков, работающих над другими частями кодовой базы. Существует еще один шаблон, позволяющий вносить изменения в наш монолит поступательно, не прибегая к ветвлению исходного кода. Шаблон "Ветвление по абстракции" (branch by abstraction) вместо этого полагается на внесение изменений в существующую кодовую базу, допуская безопасное сосуществование имплементаций друг с другом в одной и той же версии кода, не вызывая слишком большого нарушения работы.

Как он работает

Ветвление по абстракции состоит из пяти шагов:

1. Создать абстракцию для заменяемой функциональности.
2. Изменить клиентов существующей функциональности так, чтобы они использовали новую абстракцию.

---

[11] См. **http://bit.ly/2pctNfn**.

3. Создать новую имплементацию абстракции с переработанной функциональностью. В нашем случае эта новая имплементация будет вызывать новую микрослужбу.
4. Переключиться на абстракцию, для того чтобы использовать новую имплементацию.
5. Очистить абстракцию и удалить старую имплементацию.

Давайте рассмотрим эти шаги по переносу функциональности "Уведомлений" в службу, как было показано на рис. 3.4.

Шаг 1: создать абстракцию

Первая часть работы состоит в создании абстракции, представляющей взаимодействия между изменяемым кодом и элементами, вызывающими этот код, как показано на рис. 3.22. Если существующая функциональность "Уведомлений" достаточно хорошо факторизована, то эта работа просто сводится к применению рефакторизации с извлечением интерфейса в нашей среде IDE. Если нет, то вам потребуется извлечь стык, как упоминалось ранее. Это заставит вас просканировать свою кодовую базу на предмет вызовов, выполняемых к API-интерфейсам, которые отправляют электронные письма, SMS-сообщения или любой другой механизм уведомлений, который у вас есть. Отыскание этого кода и создание абстракции, которую другой код использует, является обязательным шагом.

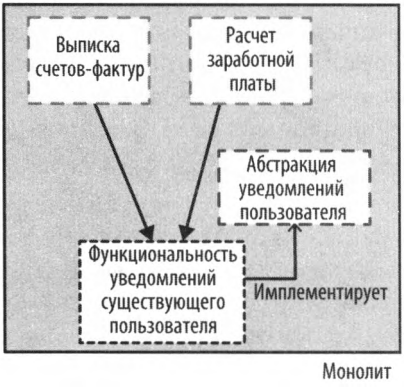


Рис. 3.22. Шаг 1: создание абстракции

Шаг 2: использовать абстракцию

Теперь, когда наша абстракция создана, нам нужно рефакторизовать существующих клиентов функциональности "Уведомления", для того чтобы использовать эту новую точку абстракции, как мы видим на рис. 3.23. Возможно, что рефакторизация с извлечением интерфейса уже сделала это за нас автоматически — но, по моему опыту, чаще всего этот процесс должен быть поступательным, включающий ручное отслеживание входящих вызовов затрагиваемой функциональности. Самое приятное здесь то, что эти изменения являются малыми и поступательными; их легко делать малыми шагами, не оказывая слишком большого влияния на существующий код. В этой точке не должно быть никаких функциональных изменений в поведении системы.

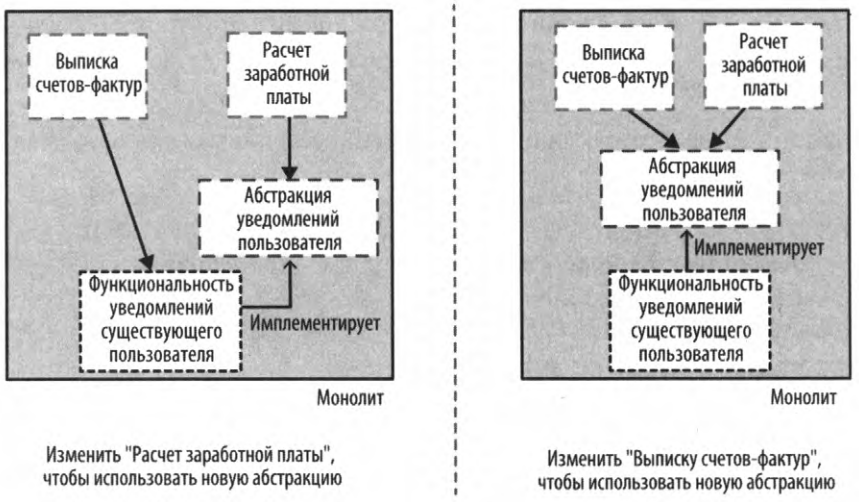


Рис. 3.23. Шаг 2: изменить существующих клиентов так, чтобы они использовали новую абстракцию

Шаг 3: создать новую имплементацию

Когда новая абстракция находится на своем месте, можно начать работу над новой имплементацией вызова службы. Внутри монолита имплементация функциональности "Уведомления" будет представлять собой главным образом просто клиента, который вызывает внешнюю службу, как показано на рис. 3.24. Подавляющая часть функциональности будет находиться в самой службе.

Ключевая вещь в этой точке состоит в том, что, хотя мы имеем две имплементации абстракции в кодовой базе одновременно, только одна имплементация в настоящее время активна в системе. До тех пор пока мы не будем довольны тем, что наша новая имплементация вызова службы готова к отправке в "прямой эфир", она практически находится в спячке. В то время как мы работаем над имплементацией всей эквивалентной функциональности в нашей новой службе, наша новая имплемента-

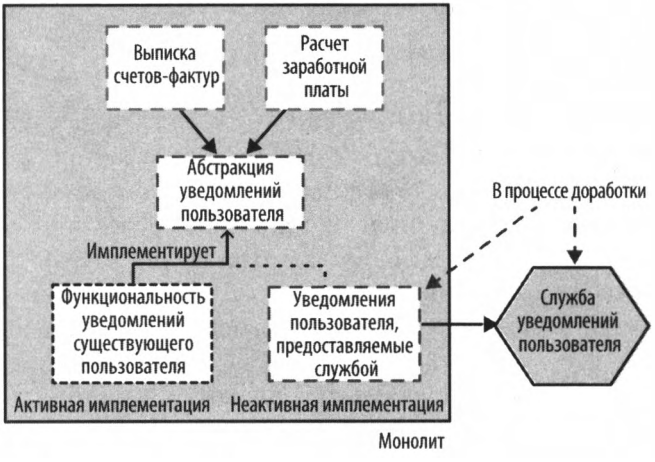


Рис. 3.24. Шаг 3: Создать новую имплементацию абстракции

ция абстракции будет возвращать ошибки `Not Implemented` (Не имплементировано). Это, конечно, не мешает писать тесты для написанной функциональности, и это одна из выгод от интегрирования данной работы как можно раньше.

В ходе этого процесса мы также развертываем в производство службу "Уведомление пользователя", находящуюся в процессе доработки, как это было сделано с шаблоном "Фикус-удавка". То, что она не закончена, не играет роли, поскольку в этой точке имплементация абстракции уведомлений не находится в "прямом эфире", указанная служба фактически не вызывается. Но мы можем ее развернуть, протестировать прямо на месте и верифицировать правильную работу тех частей функциональности, которые мы имплементировали.

Бывает, что эта фаза длится достаточно долго. Джез Хамбл (Jez Humble) подробно описывает применение шаблона "Ветвление по абстракции"[12] для мигрирования слоя постоянства базы данных, используемого в приложении GoCD по непрерывной доставке (в то время именуемом Cruise). Переход от iBatis к Hibernate продолжался несколько месяцев, в течение которых приложение по-прежнему продолжало поставляться клиентам дважды в неделю.

Шаг 4: переключить имплементацию

Как только мы будем довольны тем, что наша новая имплементация работает правильно, мы переключим нашу точку абстракции так, чтобы новая имплементация стала активной, а старая функциональность больше не использовалась, как показано на рис. 3.25.

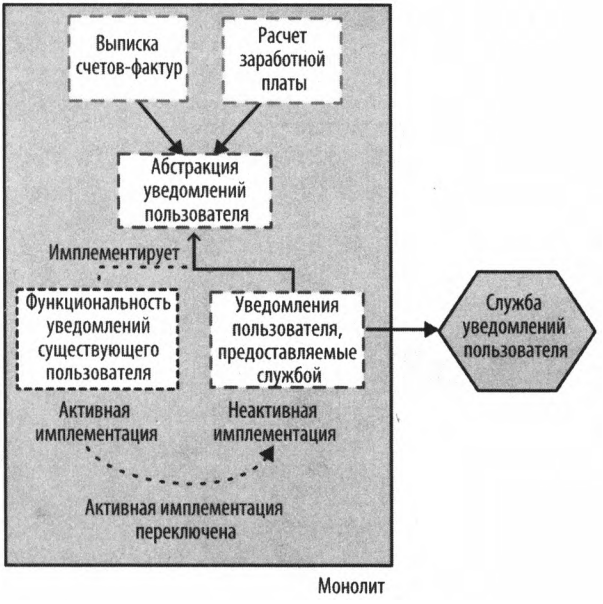


Рис. 3.25. Шаг 4: переключиться на активную имплементацию для использования новой микрослужбы

[12] См. **http://bit.ly/2p95lv7**.

В идеале, как и в случае с "фикусом-удавкой", мы хотели бы использовать переключающий механизм, который легко принимает одно из двух состояний. Это позволило бы быстро переключаться назад на старую функциональность при возникновении проблем. Общепринятое решение — использование реле с двумя состояниями (флажка). На рис. 3.26 мы видим, что указанные реле имплементируются с помощью конфигурационного файла, позволяя нам менять действующую имплементацию без необходимости изменять код. Если вы хотите узнать о реле функций

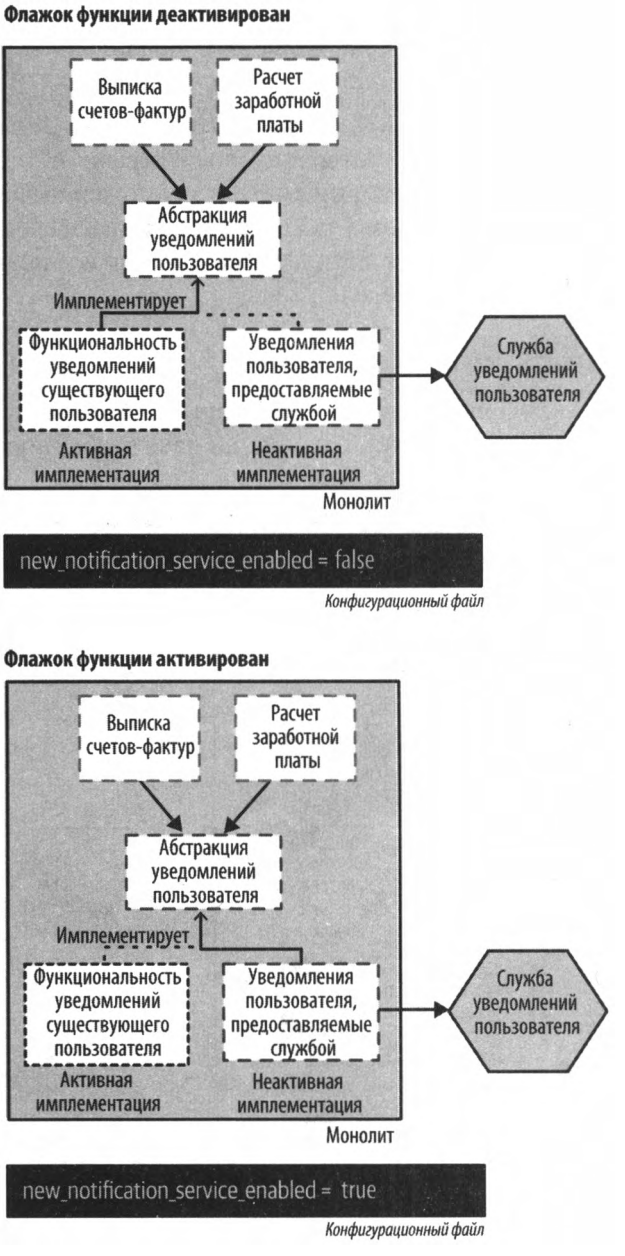


Рис. 3.26. Шаг 5: использование реле функций для переключения между имплементациями

больше и о том, как их имплементировать, то у Пита Ходжсона (Pete Hodgson) есть отличная тематическая подборка.

На данном этапе мы имеем две имплементации одной и той же абстракции, которые, как мы надеемся, должны быть функционально эквивалентными. Для верификации эквивалентности можно задействовать тесты, но как вариант, мы можем применить обе имплементации в производстве, тем самым обеспечивая дополнительную верификацию. Эта идея рассматривается далее в разделе "Шаблон: параллельное выполнение".

Шаг 5: очистка

Теперь, когда наша новая микрослужба обеспечивает пользователям все уведомления, мы можем обратиться к очистке после себя. В этой точке наша старая функциональность "Уведомлений пользователя" больше не используется, поэтому оче-

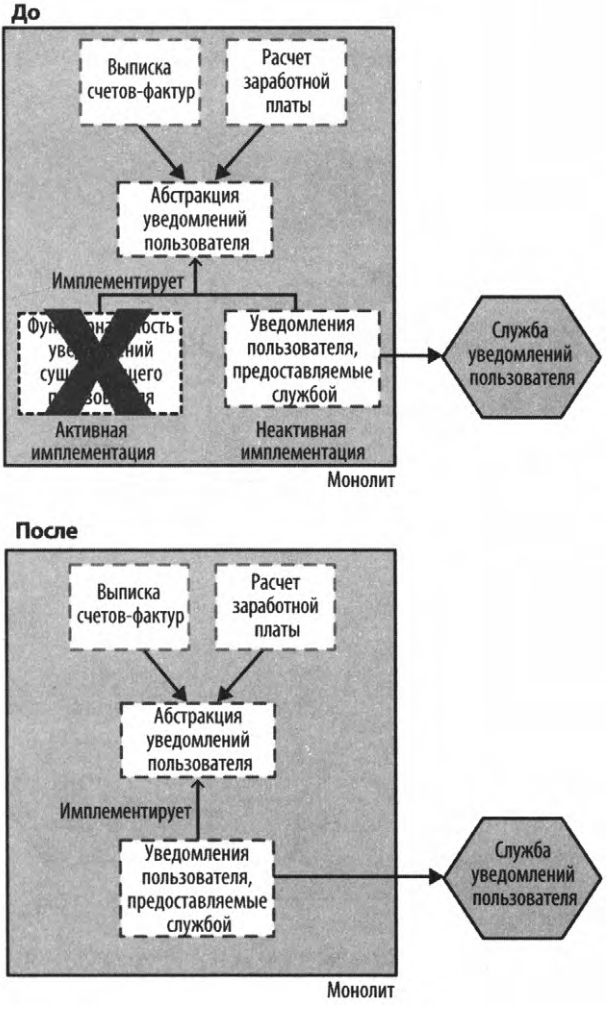


Рис. 3.27. Шаг 6: удалить старую имплементацию

видным шагом будет ее удалить, как показано на рис. 3.27. Мы начинаем сжимать монолит!

Во время удаления старой имплементации также имеет смысл удалить любое переключение флажков функций, которые мы могли имплементировать. Одна из реальных проблем, связанных с использованием флажков функций, состоит в том, что старые флажки разбросаны повсюду — не делайте этого! Для поддержания простоты удалите флажки, которые вам больше не нужны.

Наконец, когда старая имплементация убрана, у нас есть возможность удалить саму точку абстракции, как показано на рис. 3.28. Однако существует возможность, что создание абстракции улучшило кодовую базу до такой степени, что вы предпочли бы оставить ее на месте. Если это нечто такое же простое, как интерфейс, то его сохранение будет минимально влиять на существующую кодовую базу.

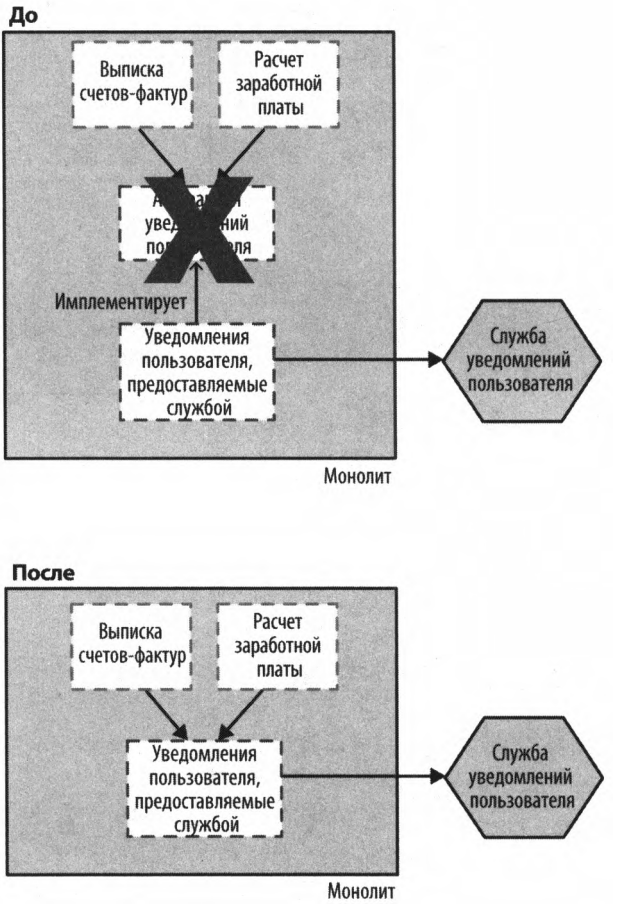


Рис. 3.28. Шаг 7: (необязательный) удалить точку абстракции

В качестве механизма отката

Возможность переключаться назад к предыдущей имплементации, если новая служба не ведет себя как надо, полезна, но есть ли способ делать это автоматически? Стив Смит (Steve Smith) подробно описывает вариацию шаблона "Ветвление по абстракции", именуемую верификацией ветвления по абстракции (verify branch by abstraction)[13], которая также имплементирует шаг верификации в "прямом эфире"; мы видим пример этого шаблона на рис. 3.29. Идея состоит в том, что если вызовы новой имплементации не срабатывают, то вместо нее можно использовать старую имплементацию.

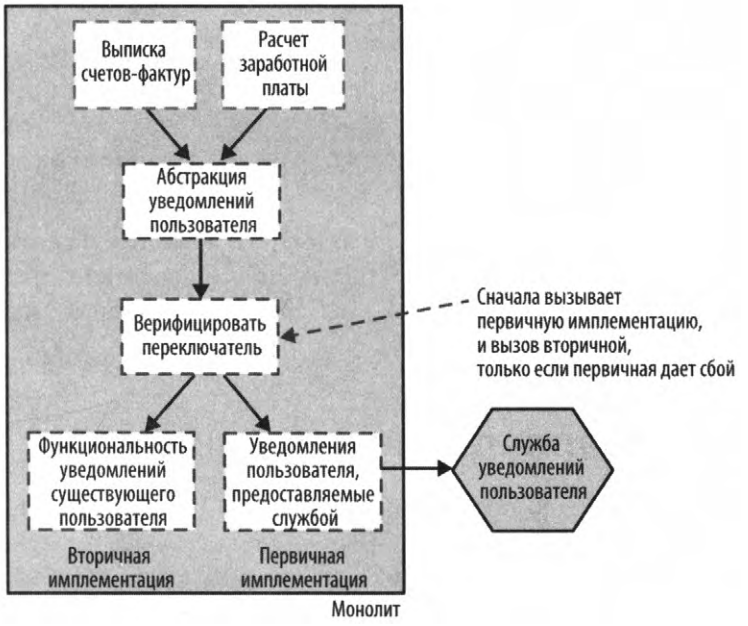


Рис. 3.29. Верификация ветвления по абстракции

Указанная вариация явно добавляет некоторую сложность не только с точки зрения кода, но и с точки зрения рассуждений о системе. В практическом плане обе имплементации в любой момент времени могут быть активными, что затрудняет понимание поведения системы. Если две имплементации так или иначе поддерживают внутреннее состояние, то нам также придется подумать о согласованности данных[14]. Хотя согласованность данных является проблемой в любой ситуации, когда мы переключаемся между имплементациями, шаблон "Верификация ветвления по абстракции" позволяет переключаться туда-сюда между имплементациями, опираясь на запрос по запросу (request-by-request). Это означает, что вам понадобится

[13] См. **http://bit.ly/2mLVevz**.

[14] Для справки: согласованность данных (data consistency), также консистентность, когерентность, означает, что значения данных одинаковы для всех экземпляров приложения — *Пер.*

согласованный совместный набор данных, к которому могут обращаться обе имплементации.

Мы рассмотрим эту идею подробнее чуть позже, когда обратимся к более обобщенному шаблону "Параллельное выполнение".

Где его использовать

Ветвление по абстракции — это довольно универсальный шаблон, полезный в любой ситуации, когда изменения в существующей кодовой базе, вероятно, потребуют времени на их исполнение, но при желании избежать слишком больших помех для ваших коллег. На мой взгляд, этот вариант оптимальнее, чем использование долгоживущих ветвей кода практически во всех обстоятельствах.

Для миграции на архитектуру, основанную на микрослужбах, я почти всегда сперва обращаюсь к использованию шаблона "Фикус-удавка", т. к. он проще во многих отношениях. Однако существует несколько ситуаций, как здесь было с уведомлениями, где это просто невозможно.

Описанный шаблон также исходит из того, что вы можете изменить код существующей системы. Если же по какой-либо причине это сделать невозможно, то вам придется рассмотреть другие варианты, некоторые из которых мы рассмотрим в остальной части этой главы.

Шаблон: "Параллельное выполнение"

Прежде чем развернуть свою новую имплементацию, предстоит выполнить огромный массив работы по ее тестированию. Вы будете делать все возможное для того, чтобы предрелизная верификация вашей новой микрослужбы была выполнена в обстановке, максимально приближенной к производственной, в рамках обычного процесса тестирования, но мы все понимаем, что не всегда возможно продумать каждый сценарий, который может произойти в производственной среде. Однако у нас имеются и другие методы.

И шаблон "Фикус-удавка", и шаблон "Ветвление по абстракции" позволяют старым и новым имплементациям одной и той же функциональности одновременно сосуществовать в производстве. В типичной ситуации оба этих метода позволяют нам исполнять либо старую имплементацию в монолите, либо новое решение на основе микрослужб. Для снижения риска переключения на новую имплементацию, основанную на службах, эти методы дают возможность быстро переключаться назад к предыдущей имплементации.

Во время использования параллельного выполнения (parallel run) вместо вызова старой или новой имплементации мы вызываем *обе*, что позволяет нам сравнивать результаты с целью обеспечения их эквивалентности. Несмотря на вызов обеих имплементаций, в любой момент времени только одна из них считается источником истины. В типичной ситуации старая имплементация считается источником исти-

ны, до тех пор пока продолжающаяся верификация не покажет, что мы можем доверять новой имплементации.

Этот шаблон использовался в разных формах на протяжении десятилетий, хотя, как правило, он предназначен для параллельного выполнения двух систем. Могу утверждать, что этот шаблон столь же полезен внутри отдельной системы во время сравнения двух имплементаций одной и той же функциональности.

Этот метод служит для верификации не только того, что наша новая имплементация дает те же отклики, что и существующая имплементация, но и того, что она также работает в рамках приемлемых нефункциональных параметров. Например, достаточно ли скоро откликается наша новая служба? Не наблюдаем ли мы слишком много тайм-аутов?

Пример: сравнение ценообразования кредитных деривативов

Много лет назад я участвовал в проекте по изменению платформы, используемой для расчетов типа финансового продукта, именуемого кредитными деривативами. Банк, в котором я работал, должен был обеспечивать, чтобы различные деривативы, которые он предлагал, были для них разумной сделкой. Заработаем ли мы деньги на этой сделке? Не является ли эта сделка слишком рискованной? К тому же, после выпуска деривата рыночные условия изменяются. Поэтому им также нужно было оценивать стоимость текущих сделок для обеспечения неуязвимости перед огромными убытками из-за изменившихся рыночных условий[15].

Мы почти полностью заменили существующую систему, которая выполняла эти важные расчеты. Из-за большого количества привлеченных денег и того факта, что некоторые вознаграждения людей частично основывались на стоимости заключавшихся сделок, существовала огромная степень беспокойства по поводу изменений. Мы приняли решение выполнять два набора вычислений бок о бок и проводить ежедневные сравнения результатов. События ценообразования запускались с помощью событий, которые легко дублировались таким образом, чтобы обе системы выполняли расчеты, как мы видим на рис. 3.30.

Каждое утро мы выполняли пакетную сверку результатов, а затем должны были объяснить любые изменения в результатах. На самом деле мы написали программу для проведения сверки. Мы представляли результаты в электронной таблице Excel, что позволило легко обсуждать их с экспертами банка.

Оказалось, что у нас было несколько проблем, которые нам пришлось исправить, но мы также обнаружили большее число несоответствий, вызванных дефектами

[15] Оказалось, что как отрасль мы были ужасны. Рекомендую книгу Мартина Льюиса (Martin Lewis) "Большая шишка" (The Big Short, W. W. Norton & Company, 2010) как отличный обзор роли кредитных деривативов в мировом финансовом кризисе 2007-2008 годов. Я часто с огромным сожалением оглядываюсь назад на ту небольшую роль, которую я сыграл в этой индустрии. Оказывается, незнание того, что вы делаете, и продолжение этой работы может иметь некоторые довольно катастрофические последствия.

в существующей системе. Это означало, что некоторые из отличавшихся результатов на самом деле были правильными, но мы должны были показать нашу работу (что стало намного проще из-за появления результатов в Excel). Помню, что мне приходилось садиться с аналитиками и объяснять, почему наши результаты были правильными, раскладывая вещи по полочкам, начиная с первых принципов.

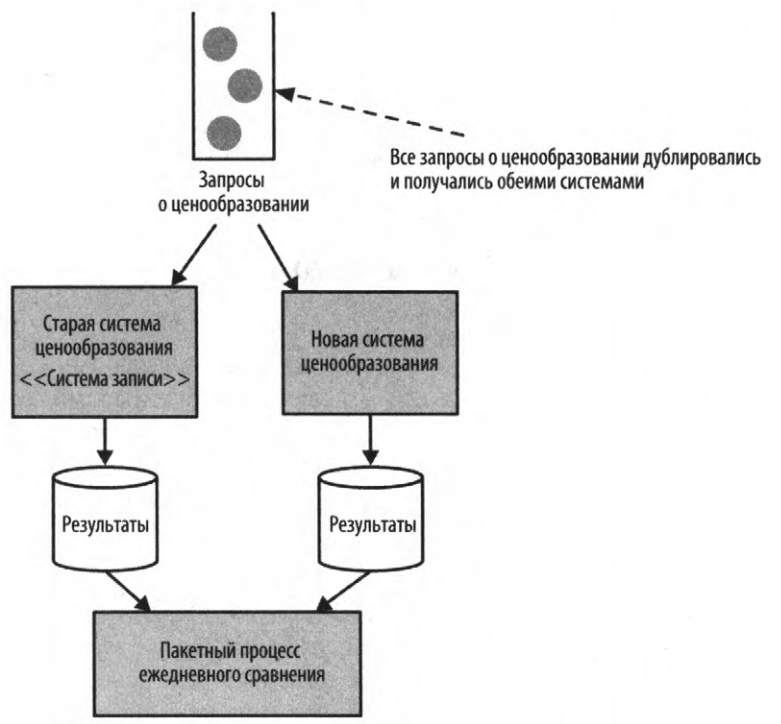


Рис. 3.30. Пример параллельного выполнения — активируются обе системы ценообразования, а результаты сравниваются в оффлайновом режиме

В конце концов, через месяц мы перешли к использованию нашей системы в качестве источника истины для расчетов, а некоторое время спустя мы вывели старую систему из эксплуатации (мы держали ее еще несколько месяцев на тот случай, если понадобится провести какой-либо аудит расчетов, выполнявшихся в старой системе).

Пример: листинги компании Homegate

Как мы обсуждали ранее в разделе "Пример: FTP", компания Homegate параллельно выполняла обе свои системы импорта листингов, при этом новая микрослужба занималась импортом листингов, которые затем сравнивались с существующим монолитом. Отдельная закачка по FTP со стороны клиента приводила к запуску обеих систем. После того как они получили подтверждение о том, что новая микрослужба ведет себя эквивалентно, импорт по FTP в старом монолите был деактивирован.

N-вариантное программирование

Можно утверждать, что вариация параллельного выполнения существует в некоторых системах управления с особыми требованиями к безопасности, таких, как воздушные судна с электродистанционным управлением. Вместо того чтобы полагаться на механическое управление, авиалайнеры все больше опираются на цифровые системы управления. Когда для управления рулем направления вместо натягивания троса пилот использует элементы управления, воздушное судно с электродистанционным управлением посылает входные данные в системы управления, которые решают, насколько следует повернуть руль направления. В обязанностях этих систем управления находится интерпретация сигналов, которые им посылают, и выполнение надлежащих действий.

Очевидно, что дефект в этих системах управления чрезвычайно опасен. Для компенсации влияния дефектов в некоторых ситуациях параллельно используются несколько имплементаций одной и той же функциональности. Сигналы посылаются всем имплементациям одной и той же подсистемы, которые затем посылают свой отклик. Эти результаты сравниваются, и "правильный" отбирается, как правило, путем поиска кворума среди участников. Это метод носит название "N-вариантное программирование"[16].

Конечная цель такого подхода не в том, чтобы заменить одну из имплементаций, в отличие от других шаблонов, которые мы рассмотрели в этой главе, а, наоборот, альтернативные имплементации будут продолжать существовать рядом друг с другом, в надежде, что альтернативные имплементации уменьшат влияние дефекта в любой данной подсистеме.

Методы верификации

При параллельном выполнении мы хотим сравнить функциональную эквивалентность двух имплементаций. Если взять рассмотренный ранее пример ценообразователя кредитного деривата, то мы можем рассматривать обе версии как функции — при наличии одинаковых входов мы ожидаем одинаковые выходы. Но мы также можем (и должны) подтвердить правильность и нефункциональных аспектов. Бывает, что вызовы, выполняемые через контуры сети, приводят к значительным задержкам и становятся причиной потери запросов из-за тайм-аутов, разделов и т. п. Поэтому наш верификационный процесс должен также распространяться на обеспечение своевременного завершения вызовов новой микрослужбы с приемлемой частотой сбоев.

Использование "шпионов"

В предыдущем примере с "Уведомлениями" мы не хотели бы отправлять электронное письмо клиенту дважды. В такой ситуации "шпион" (spy) окажется весьма кстати. Взятый из модульного тестирования указанный структурный шаблон подменяет часть функциональности и позволяет нам выполнять верификацию после

---

[16] См. Лайминг Чен (Liming Chen) и Альгирдас Авициенс (Algirdas Avizienis) "N-вариантное программирование: отказоустойчивый подход к надежности операций на основе программно-информационного обеспечения (N-Version Programming: A Fault-Tolerance Approach to reliability of Software Operation), опубликованный в сборнике 25-го Международного симпозиума по отказоустойчивому программированию (the Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995).

того, как некоторые вещи были сделаны. "Шпион" встает и заменяет часть функциональности, заглушая ее.

Таким образом, для нашей функциональности "Уведомлений" мы могли бы с помощью "шпиона" заменить низкоуровневый код, который служит для фактической отправки электронной почты, как показано на рис. 3.31. Наша новая служба "Уведомления" будет использовать этого "шпиона" во время фазы параллельного выполнения, позволяя нам проверять, что этот побочный эффект (отправка электронной почты) будет вызван, когда вызов `sendNotification` будет получен службой.

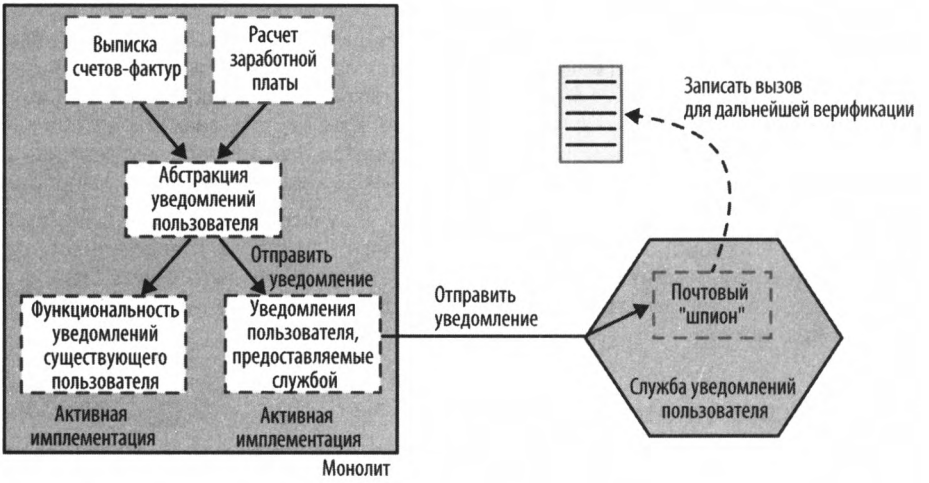


Рис. 3.31. Использование "шпиона" для проверки отправки электронных писем во время параллельного выполнения

Обратите внимание, что мы могли бы решить использовать "шпиона" внутри монолита и вообще избежать вызова службы нашим кодом `RemoteNotification`. Однако, скорее всего, вы на самом деле хотите учесть влияние дистанционных (удаленных, remote) вызовов — для понимания того, не являются ли причиной проблем таймауты, сбои или общая задержка в нашей новой службе "Уведомлений".

Дополнительная сложность здесь заключается в том, что "шпион" работает в отдельном процессе, который усложнится, когда выполняется верификационный процесс. Если бы мы хотели, чтобы это происходило в "прямом эфире", в рамках исходного запроса, то нам, вероятно, нужно было бы выставить наружу методы, определенные на службе "Уведомления", чтобы разрешить выполнять верификацию после отправки первоначального вызова в нашу службу "Уведомления". Это будет представлять значительный объем работы, и во многих случаях верификация в "прямом эфире" не требуется. Более вероятной моделью верификации внепроцессных "шпионов" будет регистрация взаимодействий, которая позволяет проводить верификацию внеполосно, возможно, на ежедневной основе. Очевидно, что если бы мы все-таки задействовали "шпиона" для замены вызова службы "Уведомлений", то проверка стала бы проще, но зато тестировать придется меньше! Чем больше функциональности вы заменяете "шпионом", тем меньше функциональности тестируется фактически.

Библиотека Scientist хостинга GitHub

Библиотека Scientist[17] хостинга GitHub — это примечательная библиотека, которая помогает имплементировать этот шаблон на уровне кода. Написанная на Ruby указанная библиотека позволяет выполнять имплементации бок о бок и собирать информацию о новой имплементации, помогая вам понять, насколько правильно она работает. Я не использовал ее сам, но вижу, как наличие подобного рода библиотеки действительно поможет в верификации вашей новой функциональности, основанной на микрослужбах, относительно существующей системы. Теперь указанная библиотека портирована на многочисленные языки, включая Java, .NET, Python, Node.JS и многие другие, помимо упомянутых.

"Темный" запуск и выпуск "канареечных" релизов

Стоит отметить, что параллельное выполнение отличается от того, что традиционно именуется как выпуск "канареечного" релиза. "Канареечный" релиз (canary release) связан с направлением некоторого подмножества ваших пользователей к новой функциональности, при этом подавляющая часть пользователей видит старую имплементацию. Идея состоит в том, что если новая система имеет проблему, то только подмножество запросов подвержено влиянию этой проблемы. При параллельном выполнении мы вызываем обе имплементации.

Еще один родственный метод называется "темным" запуском (dark launching). При "темном" запуске вы развертываете новую функциональность и ее тестируете, но новая функциональность для ваших пользователей невидима. Поэтому параллельное выполнение является способом имплементации "темного" запуска, поскольку "новая" функциональность практически невидима для ваших пользователей до тех пор, пока вы не переключитесь на систему, которая работает в "прямом эфире".

"Темный" запуск, параллельные выполнения и выпуск "канареечных" релизов — все эти методы можно использовать для верификации того, что наша новая функциональность работает правильно, и для уменьшения влияния, если окажется, что это не так. Все эти методы попадают под так называемое знамя "прогрессивной доставки" — зонтичного термина, придуманного Джеймсом Гавернером (James Governor)[18] для описания методов, помогающих контролировать то, как софт развертывается для ваших пользователей более нюансированным способом, позволяя вам выпускать софт быстрее, при этом верифицируя его эффективность и уменьшая влияние проблем, если они возникают.

Где его использовать

Имплементация параллельного выполнения редко оказывается тривиальным делом и обычно зарезервирована для тех случаев, когда считается, что изменяемая функциональность находится под высокой степенью риска. В *главе 4* мы рассмотрим

---

[17] См. **https://github.com/github/scientist**.

[18] См. **http://bit.ly/2lZjrxK**.

пример использования этого шаблона для медицинской документации. Я бы определенно был достаточно избирательным в отношении того, где применять этот шаблон — работа по его имплементации должна обмениваться на выгоды, которые вы получаете. Я сам пользовался этим шаблоном всего один-два раза, но в этих ситуациях он был чрезвычайно полезен.

Шаблон: "Сотрудник-декоратор"

Что произойдет, если вы захотите вызвать какое-то поведение, основываясь на том, что происходит внутри монолита, но вы не способны изменить сам монолит? Шаблон "Сотрудник-декоратор" (decorating collaborator) окажет здесь большую помощь. Широко известный структурный шаблон "Декоратор" позволяет прикреплять новую функциональность к чему-либо без того, чтобы лежащая в основании вещь что-то об этом "знала". Мы собираемся использовать декоратор, для того чтобы сделать вид, что наш монолит делает вызовы нашей службы напрямую, даже если мы на самом деле не изменили лежащий в основании монолит.

Вместо перехвата этих вызовов до того, как они достигнут монолита, мы даем вызову выполняться как обычно. Затем, основываясь на результате этого вызова, мы обращаемся к нашим внешним микрослужбам через любой механизм сотрудничества, который мы выберем. Давайте подробно рассмотрим эту идею на примере компании Music Corp.

Пример: программа лояльности

Music Corp всецело строится на заботе о наших клиентах! Мы хотим добавить им возможность зарабатывать баллы на основе размещаемых ими заказов, но наша текущая функциональность размещения заказов достаточно сложна, и мы предпочли бы не менять ее прямо сейчас. Поэтому функциональность размещения заказов останется в существующем монолите, но мы будем использовать прокси-селектор для перехвата этих вызовов и на основе результата решать, сколько баллов выставлять, как показано на рис. 3.32.

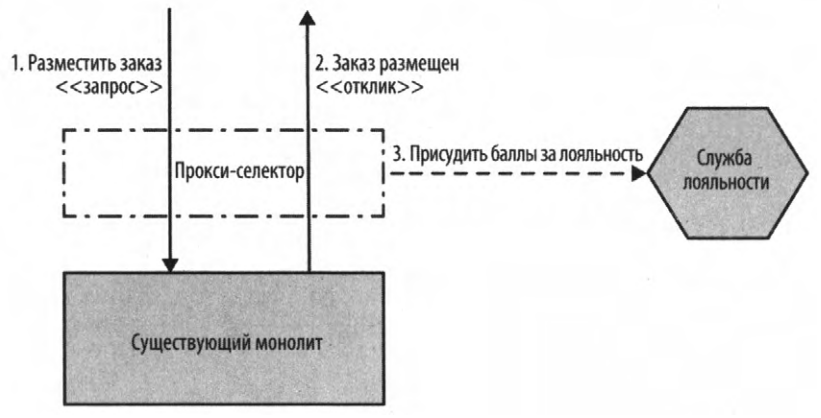


Рис. 3.32. Когда заказ успешно размещен, наш прокси-селектор обращается к службе "Лояльности" для начисления баллов клиенту

С шаблоном "Фикус-удавка" прокси-селектор был довольно упрощенным. Теперь наш прокси-селектор имеет возможность воплотить еще несколько "умов". Он должен делать свои собственные вызовы новой микрослужбы и туннелировать отклики назад клиенту. Как и прежде, приглядывайте за сложностью, которая "сидит" в прокси-селекторе. Чем больше кода вы начинаете здесь добавлять, тем больше он становится микрослужбой сам по себе, хотя и в техническом плане, со всеми трудностями, которые мы обсуждали ранее.

Еще одна потенциальная трудность заключается в том, что для обеспечения возможности делать вызов микрослужбы нам нужен достаточный объем информации из входящего запроса. Например, если мы хотим присудить баллы, основываясь на стоимости заказа, но стоимость заказа не ясна ни из запроса в "Размещении заказа", ни из отклика оттуда, то нам потребуется поиск дополнительной информации, возможно, обратный вызов в монолит для извлечения нужной информации, как показано на рис. 3.33.

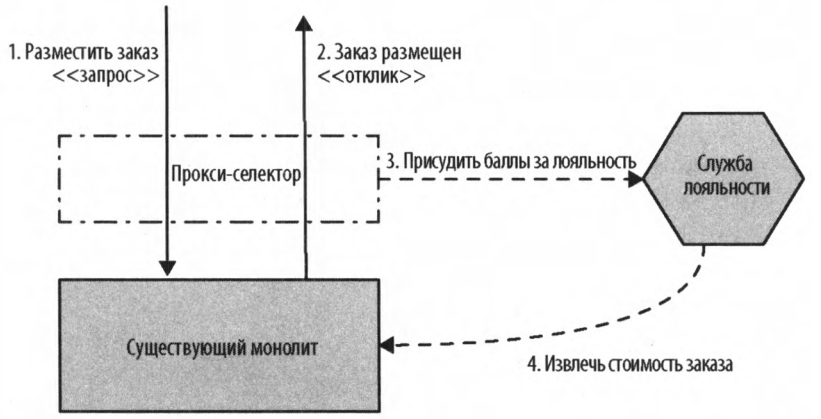


Рис. 3.33. Наша служба "Лояльности" должна загрузить дополнительные детали заказа для выяснения того, сколько баллов присуждать

Поскольку этот вызов генерирует дополнительную нагрузку и, возможно, вводит циклическую зависимость, было бы лучше изменить монолит для обеспечения необходимой информации, после того как размещение заказа будет завершено. Это, однако, потребует либо изменения кода монолита, либо, возможно, использования более инвазивного метода, такого, как захват изменений в данных.

Где его использовать

При сохранении простоты этот подход является наиболее элегантным и наименее сопряженным, чем захват изменений в данных. Описанный шаблон лучше всего работает там, где требуемая информация извлекается из входящего запроса или отклика из монолита. Там, где для выполнения правильных вызовов вашей новой службы требуется больше информации, его имплементация становится сложнее и запутаннее. По моим внутренним ощущениям, если запрос в монолит и отклик из

него не содержит нужной вам информации, то хорошенько подумайте, прежде чем использовать этот шаблон.

Шаблон: "Захват изменений в данных"

В рамках шаблона "Захват изменений в данных" (change data capture), вместо того чтобы пытаться перехватывать и действовать по вызовам в монолит, мы реагируем на изменения, вносимые в хранилище данных. Для правильной работы шаблона необходимо, чтобы опорная система захвата была сопряжена с хранилищем данных монолита. В этом по-настоящему неизбежная трудность с указанным шаблоном.

Пример: выпуск карточек лояльности

Мы хотим интегрировать немного функциональности с целью распечатки карточек лояльности для наших пользователей, когда они регистрируются. В настоящее время учетная запись лояльности создается при регистрации клиента. Как мы видим на рис. 3.34, когда регистрация возвращается из монолита, мы знаем только, что клиент был успешно зарегистрирован. Чтобы мы могли напечатать карточку, нам нужно больше сведений о клиенте, что затрудняет вставку этого поведения выше по течению, возможно, с помощью "сотрудника-декоратора". В точке, где вызов возвращается, нам нужно будет сделать дополнительные запросы в монолит, чтобы извлечь другую необходимую нам информацию, и эта информация может и не быть выставлена наружу через API.

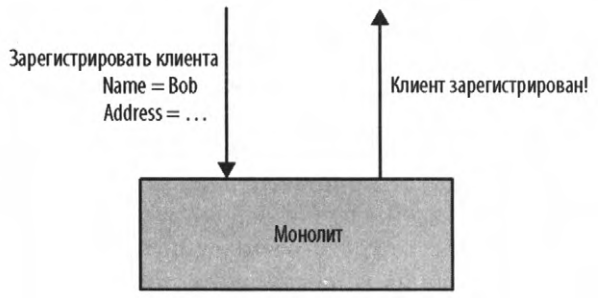


Рис. 3.34. Наш монолит делится не очень большой информацией, когда клиент зарегистрирован

Вместо этого мы решили использовать захват изменений в данных. Мы засекаем любые вставки в таблицу Loyalty_Accounts (Учетные записи лояльности), и при наступлении вставки вызываем нашу новую службу "Печати карточек лояльности", как показано на рис. 3.35. В этой конкретной ситуации мы решаем запустить событие "Учетная запись создана". Наш процесс печати лучше всего работает в пакетном режиме, позволяя накапливать список печатных заданий, которые будут выполнены в нашем брокере сообщений.

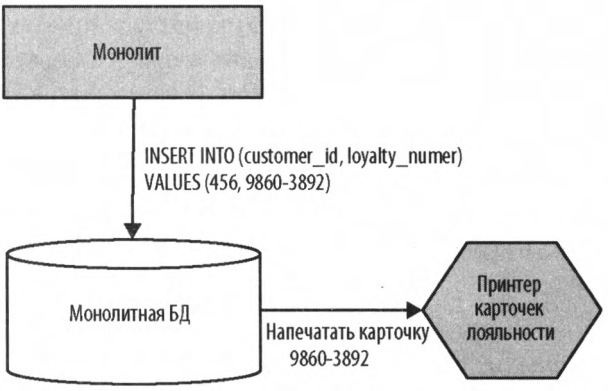


Рис. 3.35. Применение захвата изменений в данных для вызова новой службы печати

Имплементация захвата изменений в данных

Для имплементации захвата изменений в данных используются разнообразные методы, каждый из которых имеет разные компромиссы с точки зрения сложности, надежности и своевременности. Давайте рассмотрим несколько вариантов.

Триггеры базы данных

Большинство реляционных баз данных позволяют инициировать настраиваемое под свои нужды поведение во время изменения данных. То, как именно эти триггеры определены и что они могут вызывать, варьируется от системы к системе, но все современные реляционные базы данных так или иначе их поддерживают. В примере на рис. 3.36 наша служба вызывается всякий раз, когда выполняется вставка в таблицу Loyalty_Accounts.

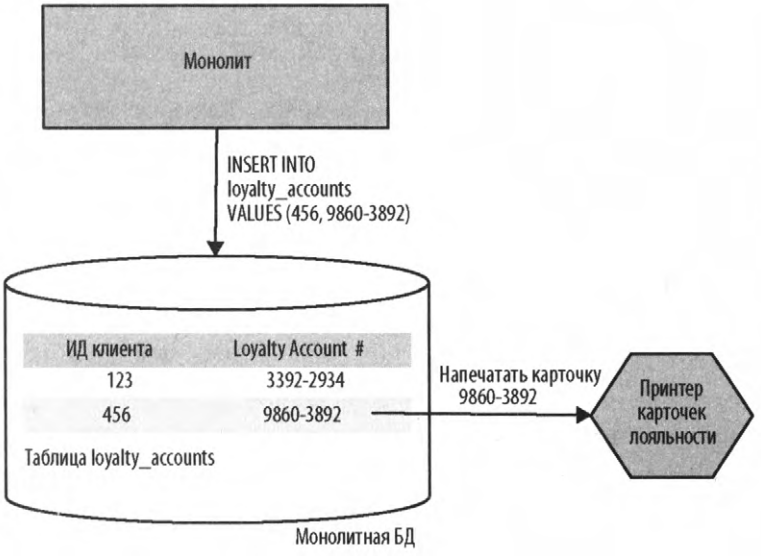


Рис. 3.36. Использование триггера базы данных для вызова микрослужбы во время вставки данных

Триггеры должны быть инсталлированы в саму базу данных, как и любая другая хранимая процедура. Также могут быть ограничения относительно того, что эти триггеры делают, хотя, по крайней мере, в Oracle они вполне себе довольны тем, что вы вызываете веб-службы или настроенный под свои нужды код на Java.

На первый взгляд, может показаться, что сделать это довольно просто. Нет необходимости иметь какой-либо другой софт, не нужно вводить какие-либо новые технологии. Однако, как и хранимые процедуры, бывает, что триггеры становятся "скользким склоном".

Мой друг, Рэнди Шоуп (Randy Shoup) однажды высказался как-то так: "Не страшно иметь один-два триггера базы данных. Ужасно, когда из них строится целая система". И нередко данная проблема сопряжена именно с триггерами баз данных. Чем их у вас больше, тем труднее понять, как ваша система на самом деле работает. Эта трудность часто связана с инструментарием и контролем со стороны триггеров за изменениями — стоит задействовать их слишком много, и ваше приложение примет вид сооружения в стиле барокко.

Поэтому, если вы собираетесь их использовать, то делайте это очень экономно.

Опросники журналов транзакций

Внутри большинства баз данных, конечно же, всех магистральных транзакционных баз данных, существует журнал транзакций. Обычно это файл, в который заносится запись обо всех внесенных изменениях. В типичной ситуации для захвата изменений в данных указанный журнал транзакций задействуется со стороны наиболее изощренного инструментария.

Эти системы работают как отдельный процесс, и их взаимодействие с существующей базой данных осуществляется только через журнал транзакций, как показано на рис. 3.37. Здесь стоит отметить, что в журнале транзакций будут отображаться только зафиксированные транзакции (что резонно).

Эти инструменты потребуют понимания формата опорного журнала транзакций, который в типичной ситуации варьируется в разных типах баз данных. В связи с этим, точный перечень инструментов у вас будет зависеть от того, какую базу данных вы используете. В этом пространстве существует огромная масса инструментов, хотя многие из них служат для поддержки репликации данных. Существует также ряд решений, предназначенных для отображения изменений журнале транзакций в сообщения, которые будут помещены в брокер сообщений; это бывает очень полезно, если ваша микрослужба по своей природе асинхронная.

Несмотря на ограничения, рассмотренное решение во многих отношениях является самым изящным для имплементации захвата изменений в данных. Сам журнал транзакций показывает изменения только в базовых данных, поэтому вас не беспокоит выяснение того, что изменилось. Инструментарий работает вне самой базы данных и запускает реплику журнала транзакций, поэтому у вас будет меньше проблем, связанных с сопряженностью или конкуренцией.

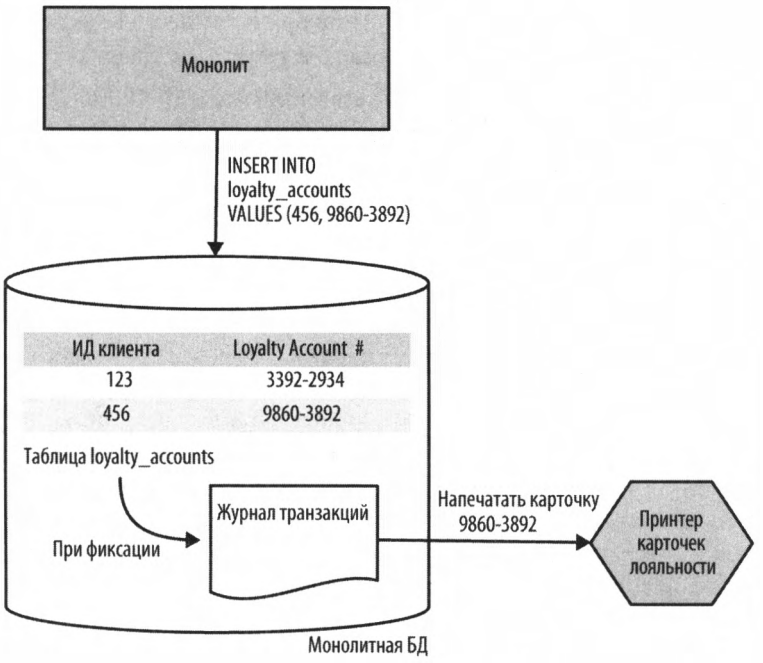


Рис. 3.37. Система захвата изменений в данных, использующая опорный журнал транзакций

Пакетный копировальщик дельты

Вероятно, наиболее упрощенный подход — написание программы, которая на регулярной основе сканирует затрагиваемую базу данных на предмет того, какие данные изменились, и копирует эти данные в место назначения. Эти задания часто выполняются с помощью таких инструментов, как `cron` или аналогичных инструментов пакетного планирования.

Главная проблема — выяснить, какие данные фактически изменились с момента последнего запуска пакетного копировальщика. Дизайн схемы данных может и не делать это очевидным. Некоторые базы данных позволяют просматривать метаданные таблиц, чтобы увидеть, когда части базы данных изменились, но этот подход далеко не универсален и будет давать вам временные метки изменений только на уровне таблицы, когда вы предпочли бы иметь информацию на уровне строк. Вы могли бы начать добавлять эти временные метки сами, но в результате объем работы значительно возрастет, а система захвата изменений в данных будет улаживать эту проблему гораздо элегантнее.

Где его использовать

"Захват изменений в данных" — полезный шаблон общего назначения, в особенности, если вам необходимо реплицировать данные (что мы рассмотрим подробнее в *главе 4*). В случае миграции на микрослужбы наиболее благоприятная зона находится там, где вам нужно реагировать на изменение в данных в вашем монолите, но

вы не способны перехватить это ни по периметру системы с помощью "удавки" или "декоратора", ни изменить лежащую в основании кодовую базу.

В общем случае я стараюсь свести использование этого шаблона к минимуму из-за трудностей, связанных с некоторыми имплементациями указанного шаблона. Триггеры базы данных обладают своими недостатками, и полномасштабные инструменты захвата изменений в данных, работающие с журналами транзакций, значительно усложняют решение. Тем не менее, если вы понимаете эти потенциальные проблемы, то описанный шаблон будет полезным инструментом в вашем распоряжении.

Резюме

Как мы увидели, широкий спектр методов позволяет выполнять поступательную декомпозицию существующих кодовых баз и помогает вам легко войти в мир микрослужб. По моему опыту, большинство людей в итоге используют сочетание подходов; редко бывает, что одна технология справляется с каждой ситуацией. Надеюсь, что к этому моменту мне удалось показать вам разнообразие подходов и дать достаточно информации, для того чтобы выяснить для себя, какие методы будут работать для вас лучше всего.

Мы пока замалчиваем одну из самых больших трудностей в мигрировании на архитектуру, основанную на микрослужбах, — данные. Мы не можем это больше откладывать! В *главе 4* мы разведаем способы мигрирования данных и разложения баз данных.

ГЛАВА 4

Декомпозиция базы данных

Как мы уже выяснили, существует масса способов извлечения функциональности из монолита в микрослужбы. Однако нам нужно обратиться к вопросу о "слоне в комнате"[1]: а именно, что делать с нашими данными? Микрослужбы работают лучше всего, когда мы практикуем сокрытие информации, что, в свою очередь, обычно приводит нас к микрослужбам, которые полностью инкапсулируют свои собственные механизмы хранения и извлечения данных. Отсюда следует вывод, что во время мигрирования на архитектуру, основанную на микрослужбах, нужно разложить базу данных нашего монолита, если мы хотим извлечь из транзита максимум.

Однако разложение базы данных — далеко не простая задача. Нам нужно учесть вопросы синхронизации данных во время транзита, логической и физической декомпозиции схемы данных, транзакционной целостности, соединений, задержки и многое другое. В этой главе мы рассмотрим эти вопросы и разведаем шаблоны, которые нам помогут.

Однако, прежде чем мы начнем разделять что-либо врозь, мы должны рассмотреть трудности — и шаблоны преодоления — для управления одной совместной базой данных.

Шаблон: "Совместная база данных"

Как уже обсуждалось в *главе 1*, мы можем думать о сопряженности в терминах доменной, временной или имплементационной сопряженности. Из трех упомянутых имплементационная сопряженность занимает нас больше всего при рассмотрении баз данных, поскольку люди часто используют одну и ту же базу данных в нескольких схемах, как показано на рис. 4.1.

На первый взгляд, существует ряд проблем, связанных с совместным использованием одной базы данных многочисленными службами. Главенствующая проблема заключается в том, что мы отказываем себе в возможности решать, что является совместным, а что скрытым — что противоречит нашему стремлению к сокрытию информации. Это означает, что будет трудно понять, какие части схемы можно изменять безопасным образом. Знать, что внешняя сторона может обращаться к вашей базе данных, — это одно, но не знать, какую часть вашей схемы она использу-

[1] То есть к очевидной проблеме, которая ставит в тупик — *Пер.*

ет, — это другое. Проблему можно смягчить с помощью проекций, которые мы обсудим в ближайшее время, но это решение не тотальное.

Еще одна трудность состоит в том, что становится неясным, кто "контролирует" данные. Где находится бизнес-логика, которая оперирует этими данными? Не "размазана" ли она теперь по всем службам? Из этого, в свою очередь, вытекает отсутствие связности бизнес-логики, как мы уже обсуждали ранее, рассматривая микрослужбу как комбинацию поведения и состояния, инкапсулирующую одну или несколько машин состояний. Если поведение, изменяющее это состояние, теперь "размазано" по всей системе, то обеспечение правильной имплементации машины состояний окажется запутанной задачей.

Если, как показано на рис. 4.1, три службы непосредственно изменяют информацию о заказе, что произойдет при отсутствии согласованного поведения между службами? Что произойдет, когда поведение нужно изменить — придется ли мне применить эти изменения ко всем этим службам? Как уже упоминалось ранее, мы стремимся к высокой связности бизнес-функциональности, и слишком часто из совместной базы данных следует обратное.

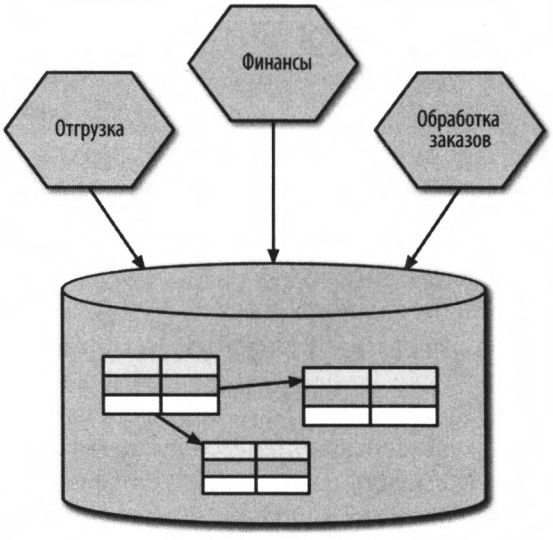


Рис. 4.1. Несколько служб напрямую обращаются к одной базе данных

Шаблоны преодоления

Хотя это мероприятие бывает трудным, разложение базы данных, для того чтобы позволить каждой микрослужбе владеть своими собственными данными, почти всегда является предпочтительным. Если это невозможно, то поможет использование шаблона "Проекция базы данных" (см. *раздел "Шаблон: проекция базы данных"*) или же принятие на вооружение шаблона "Служба обертывания базы данных" (см. *раздел "Шаблон: служба обертывания базы данных"*).

Где его использовать

По моему мнению, прямое совместное использование базы данных подходит для архитектуры на основе микрослужб только в двух ситуациях. Первая — при рассмотрении статических справочных данных, допускающих только чтение. Вскоре мы разведаем эту тему подробнее, но давайте возьмем схему, содержащую информацию о коде валюты страны, просмотровые таблицы с почтовыми кодами или индексами и тому подобное. Здесь структура данных остается очень стабильной, и контроль за изменениями в них в типичной ситуации регулируется в рамках работы по администрированию.

Второе место, где, по моему мнению, подошло бы непосредственное обращение многочисленных служб к одной и той же базе данных — там, где служба непосредственно выставляет базу данных наружу как определенную конечную точку, которая спланирована и управляется для обслуживания многочисленных потребителей. Мы рассмотрим эту идею подробнее, когда будем обсуждать шаблон "Интерфейс база-данных-как-служба" (см. *раздел "Шаблон: интерфейс база-данных-как-служба"*).

Но это невозможно сделать!

Итак, в идеале, мы хотим, чтобы наши новые службы имели свои собственные независимые схемы. Но это не то место, где мы начинаем работу с существующей монолитной системой. Означает ли это, что мы всегда должны разбивать эти схемы? Я убежден в том, что это уместно делать в большинстве ситуаций, но не всегда целесообразно делать изначально.

Иногда, как мы вскоре увидим, работа занимает слишком много времени или сопряжена с изменением особенно чувствительных частей системы. В таких случаях полезны различные шаблоны преодоления, которые, по крайней мере, остановят ухудшение ситуации, а в лучшем случае станут разумными шагами к чему-то более оптимальному в будущем.

Схемы и базы данных

В прошлом я был виновен в том, что использовал термины "база данных" и "схема" взаимозаменяемо. Это иногда приводит к путанице, поскольку в этих терминах есть некоторая двусмысленность. В техническом плане мы можем рассматривать схему как логически разделенное множество таблиц, содержащих данные, как показано на рис. 4.2. Тогда на одном ядре системы управления базами данных (СУБД, database engine) можно разместить несколько схем. В зависимости от контекста, когда люди говорят "база данных", они могут ссылаться на схему либо ядро СУБД ("база данных не работает!").

Поскольку в центре данной главы будут главным образом логические понятия баз данных, и поскольку термин "база данных" обычно используется в этом контексте, практически ссылаясь на логически изолированную схему, в этой главе я буду придерживаться этого употребления. Поэтому, когда я говорю "база данных", думайте о "логически изолированной схеме". Для краткости я опущу из наших диаграмм понятие ядра СУБД, если явно не указано иное.

Стоит отметить, что разнообразные базы данных NoSQL могут и не иметь одинакового понятия логического разделения, в особенности при работе с базами данных, предоставляемыми облачными провайдерами. Например, на AWS база данных DynamoDB имеет только понятие таблицы, а управление ролевым доступом используется для ограничения тех, кто может видеть или изменять данные. Это вызывает трудности в том, как мы думаем о логическом разделении в таких ситуациях.

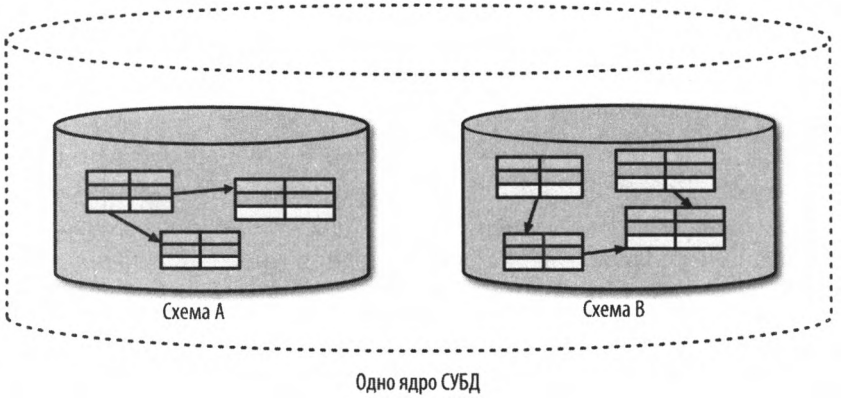


Рис. 4.2. Один экземпляр ядра СУБД вмещает многочисленные схемы, каждая из которых обеспечивает логическую изоляцию

Вы столкнетесь с проблемами с вашей текущей системой, справиться с которыми, кажется, невозможно прямо сейчас. Обратитесь к решению этой проблемы вместе с остальной частью вашей группы, чтобы каждый мог согласиться, что это действительно проблема и вы хотели бы ее решить, даже если прямо сейчас вы не видите, как. Затем убедитесь, что вы, по крайней мере, начинаете делать все правильно сейчас. Со временем проблемы, которые изначально казались непреодолимыми, станет легче решать, как только у вас появится немного новых навыков и опыта.

Шаблон: "Проекция базы данных"

В ситуации, когда нам нужен один источник данных для нескольких служб, проекция (view)[2] способна смягчить трудности, касающиеся сопряженности. При наличии проекции служба представляется схемой, которая является ограниченной проекцией из опорной схемы. Эта проекция ограничивает данные, видимые службе, скрывая информацию, к которой она не должна иметь доступа.

[2] Для справки: проекция (view), или ракурс — это набор данных, получаемый в результате сохраненного в базе данных запроса к данным, который пользователи могут выполнять точно так же, как они это делают к хранимому в базе данных объекту-коллекции. В русскоязычной литературе распространены термины "представление" и "вид", которые выглядят крайне неудачными (попробуйте перевести на русский "present a view", используя первый вариант или "a kind of view", используя второй). Во французском языке применяется термин "проекция" — *Пер.*

База данных как публичный контракт

Еще в *главе 3* я поделился своим опытом по оказанию помощи по переплатформированию существующей системы кредитных деривативов для ныне несуществующего инвестиционного банка. Мы ударили по проблеме сопряженности базы данных с размаху: нам нужно было увеличить пропускную способность системы, чтобы ускорить обратную связь с трейдерами, которые использовали систему. После небольшого анализа мы обнаружили, что узким местом в обработке были операции записи в нашу базу данных. После быстрой атаки первым темпом мы осознали, что сможем резко увеличить производительность операции записи в нашей системе, если реструктурируем схему.

Именно в этот момент мы обнаружили, что многочисленные приложения вне нашего контроля имели доступ на чтение к нашей базе данных, а в некоторых случаях доступ на чтение/запись. К сожалению, мы установили, что все эти внешние системы имели одинаковые имена пользователей и пароли, поэтому было невозможно понять, что это за пользователи и к чему они обращались. Мы подсчитали, что в этом участвовало "примерно 20" приложений, но данный результат был получен из базового анализа входящих сетевых вызовов[3].

Если каждый актор (например, человек или внешняя система) имеет разный набор учетных данных, то становится намного проще ограничивать доступ к определенным сторонам, уменьшать влияние отзыва и ротации учетных данных и лучше понимать, что делает каждый актор. Управление разными наборами учетных данных бывает болезненным, в особенности в системе на основе микрослужб, которая управляет многочисленными наборами учетных данных в расчете на одну службу. Для решения этой проблемы мне нравится применять выделенные секретные хранилища. Vault компании HashiCorp[4] — отличный инструмент в этом пространстве, поскольку он генерирует учетные данные для каждого актора для таких вещей, как базы данных, которые кратковременны и ограничены по объему.

Поэтому мы не знали, кто были эти пользователи, но нам нужно было с ними связаться. В конце концов, у кого-то возникла идея отключить совместную учетную запись, которую они использовали, и ждать, пока люди свяжутся и пойдут жалобы. Это явно было не самое лучшее решение проблемы, которое мы вообще не должны были иметь, но оно по большей части сработало. Однако вскоре мы поняли, что большинство этих приложений не проходят активного технического сопровождения, т. е. не было никаких шансов, что они будут обновлены, чтобы отразить новый дизайн схемы[5]. По сути, наша схема базы данных стала обращенным в публичное пространство контрактом, который не подлежал изменению: нам пришлось идти вперед, поддерживая эту структуру схемы.

---

[3] Когда для определения тех, кто использует вашу базу данных, вы опираетесь на анализ сети, ждите неприятностей.

[4] См. **https://www.vaultproject.io/**.

[5] Ходили слухи, что одна из систем, использующая нашу базу данных, была нейронной сетью на основе Python, которую никто не понимал, но "просто юзал".

Представляемые проекции

Наше решение состояло в том, чтобы сначала уладить те ситуации, когда внешние системы писали в нашу схему. К счастью, в нашем случае их было легко уладить. Однако для всех тех клиентов, которые хотели данные читать, мы создали выделенную схему, содержащую проекции, которые выглядели как старая схема, и вместо этого побуждали клиентов указывать на эту схему, как показано на рис. 4.3. В результате мы смогли вносить изменения в нашу собственную схему, при условии, что поддерживали указанную проекцию. Скажем просто, была задействована масса хранимых процедур.

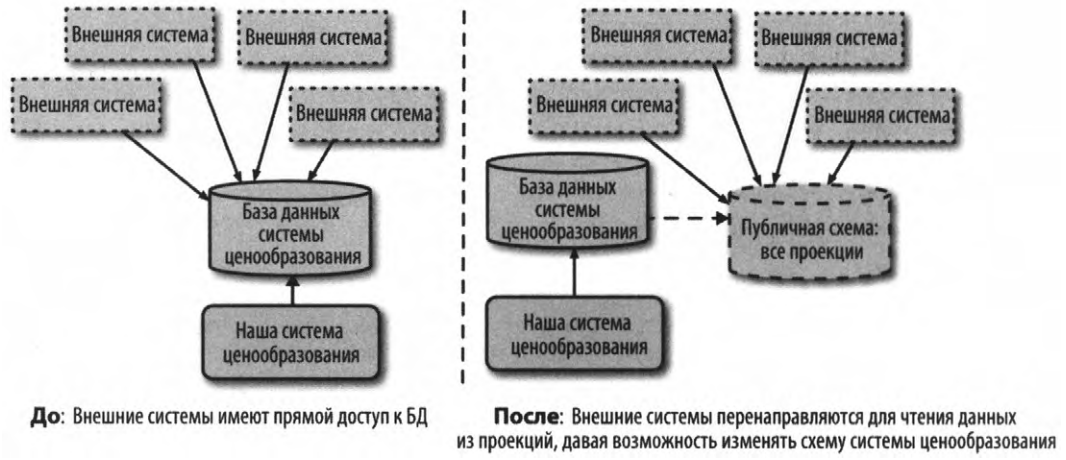


Рис. 4.3. Использование проекций дает возможность изменять базовую схему

В нашем примере инвестиционного банкинга проекция и лежащая в основе схема в итоге изрядно отличались. Разумеется, проекцию можно использовать гораздо проще, возможно, с целью скрыть фрагменты информации, которые вы не хотите делать видимыми для внешних сторон. Как простой пример на рис. 4.4, наша служба лояльности была просто списком карточек лояльности в нашей системе. В настоящее время эта информация хранится в нашей таблице клиентов в форме столбца. Поэтому мы определяем проекцию, которая выставляет наружу только отображение ИД клиента в ИД лояльности в одной единственной таблице без выставления наружу какой-либо другой информации из таблицы клиентов. Точно так же любые другие таблицы, которые находятся в базе данных монолита, полностью скрыты от службы лояльности.

Способность проекции брать только ограниченную информацию из опорного источника позволяет нам имплементировать некую форму сокрытия информации. Она дает нам контроль над тем, что является совместным, а что скрыто. Однако это решение не идеально — в таком подходе существуют ограничения.

В зависимости от природы базы данных у вас может иметься вариант создавать материализованную проекцию. При наличии материализованной проекции проекция предвычисляется — в типичной ситуации с использованием кэша. Это означает,

что чтение из проекции не должно генерировать чтение на опорной схеме, что повышает производительность. Компромисс тогда вертится вокруг того, как эта предвычисленная проекция обновляется. Может случиться так, что вы будете читать из проекции "устаревший" набор данных.

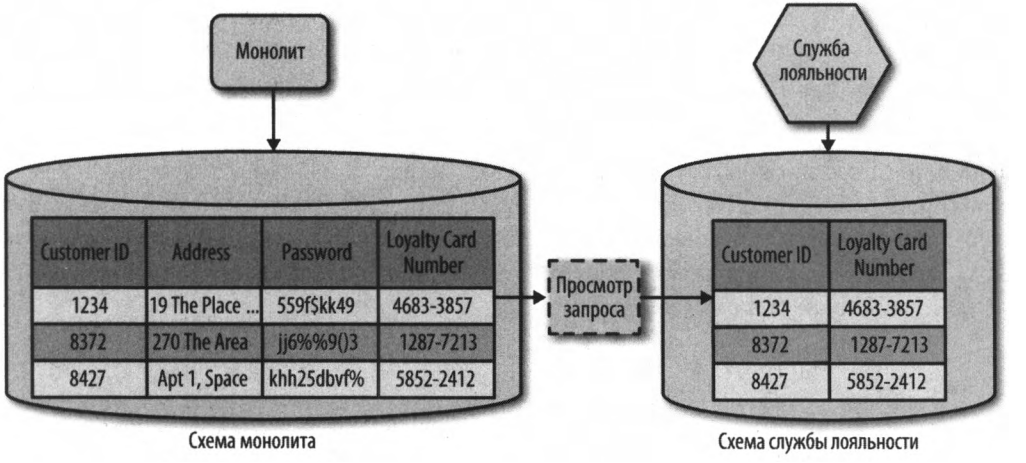


Рис. 4.4. Проекция базы данных, берущая подмножество опорной схемы

Ограничения

Способы имплементации проекций отличаются, но в типичной ситуации они являются результатом запроса. Это означает, что сама проекция доступна только для чтения. Этот факт сразу же ограничивает их полезность. Хотя проекции присущи реляционным базам данных и многие более зрелые базы данных NoSQL их поддерживают (например, Cassandra и Mongo), это делают не все. Даже если ваше ядро СУБД поддерживает проекции, скорее всего, существуют и другие ограничения, например, необходимость, чтобы исходная схема и проекция находились на одном и том же ядре СУБД. Это увеличивает сопряженность вашего физического развертывания, приводя к потенциальной единой точке сбоя.

Владение

Стоит отметить, что изменения в базовой исходной схеме требуют обновления проекции, поэтому особое внимание должно быть уделено тому, кто этой проекцией "владеет". Я предлагаю рассматривать любые опубликованные проекции базы данных как равносильные любому другому интерфейсу службы и, следовательно, как то, что должно постоянно обновляться группой, присматривающей за исходной схемой.

Где его использовать

Я использую проекцию базы данных преимущественно в ситуациях, когда считаю нецелесообразным осуществлять декомпозицию существующей монолитной схе-

мы. В идеале, вы должны стараться, если это возможно, избегать необходимости проекции, если конечная цель состоит в том, чтобы выставить эту информацию наружу через интерфейс службы. Вместо этого лучше продвигаться вперед с надлежащей декомпозицией схемы. Ограничения этого метода бывают значительными. Тем не менее, если вы чувствуете, что усилие по полной декомпозиции слишком велико, то этот шаблон будет шагом в правильном направлении.

Шаблон: "Служба обертывания базы данных"

Когда с чем-то слишком трудно справиться, имеет смысл скрыть беспорядок. С помощью шаблона "Служба обертывания базы данных" (database wrapping service) мы делаем именно это: скрываем базу данных за службой, которая действует как тонкая "обертка", перенося зависимости базы данных, которые становятся зависимостями службы, как показано на рис. 4.5.

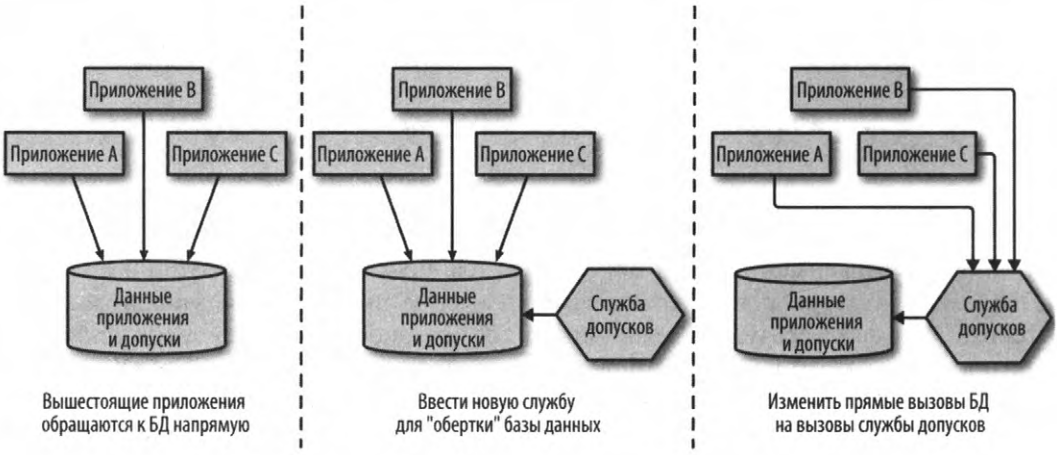


Рис. 4.5. Использование службы для "обертывания" базы данных

Несколько лет назад я работал в крупном банке в Австралии над небольшим поручением — помочь одной части организации имплементировать усовершенствованный путь к производству. В первый день мы провели несколько собеседований с ключевыми людьми, чтобы понять проблемы, с которыми они сталкиваются, и составить общий вид текущего процесса. В перерыве между собеседованиями подошел один человек и представился администратором БД этой части компании. "Пожалуйста, — сказал он, — сделайте так, чтобы они перестали помещать вещи в базу данных!".

Мы схватили по чашечке кофе, и администратор БД выложил нам проблемы. В течение примерно 30-летнего периода сформировалась система расчетно-кассового обслуживания — одна из главных "жемчужин" организации. Одна из наиболее важных частей этой системы — управление тем, что они называли "допусками". В расчетно-кассовом обслуживании управление тем, какие люди получают доступ к каким счетам, и выяснение того, что им разрешено делать с этими счетами, были

очень сложными. Чтобы дать вам понять, как выглядят эти допуски, возьмем бухгалтера, которому разрешено просматривать счета компаний *A*, *B* и *C*. По компании *B* он может переводить до $500 между счетами, а по компании *C* он может делать неограниченные переводы между счетами, но также снимать со счета до $250. Поддержание и применение этих допусков осуществлялось почти исключительно в рамках хранимых процедур базы данных. Весь доступ к данным был ограничен с помощью этой логики допусков.

По мере того как банк расширялся, а количество логики и внутреннего состояния росло, база данных начала "прогибаться" под нагрузкой. "Мы отдали в Oracle все деньги, какие только можно, но их все равно недостаточно". Опасение состояло в том, что с учетом спроецированного роста и даже в расчете на улучшение производительности аппаратного обеспечения они в итоге достигнут такого уровня, когда потребности организации превысят возможности базы данных.

По мере дальнейшего разведывания проблемы мы обсудили идею выделения частей схемы для уменьшения нагрузки. Затруднение заключалось в том, что запутанным "пучком" в середине всего была та самая система допусков. Был бы просто кошмар, если бы мы попытались его распутать, и риски, связанные с ошибками на этом участке, были огромными: сделаешь неправильный шаг, и кто-то будет заблокирован от доступа к своим счетам, или, что еще хуже, кто-то, кто не должен, получит доступ к вашим деньгам.

Мы придумали план в попытке разрешить описанную ситуацию. Мы согласились с тем, что в ближайшее время не сможем внести изменения в систему допусков, поэтому было крайне важно, чтобы мы, по меньшей мере, не усугубили проблему. Нам нужно было сделать так, чтобы люди прекратили размещать большой объем данных и поведения в схему допусков. Как только это встало бы на свое место, мы могли подумать об удалении тех частей схемы допусков, которые было легче извлечь, как мы надеялись, сократив нагрузку настолько, что опасения по поводу долгосрочной жизнеспособности будут уменьшены. В результате появилась бы некоторая передышка, для того чтобы подумать о следующих шагах.

Мы обсудили введение новой службы "Допуски", которая позволит нам "скрыть" проблемную схему. Эта служба сначала будет иметь очень мало поведения, т. к. в текущей базе данных уже имплементировано много поведения в качестве хранимых процедур. Но цель состояла в том, чтобы побудить группы, пишущие другие приложения, думать о схеме допусков как о чьей-то другой и побуждать их хранить свои собственные данные локально, как мы видим на рис. 4.6.

Как и проекции базы данных, служба "обертывания" позволяет контролировать то, что является совместным, а что скрытым. Она предоставляет потребителям интерфейс, который можно исправить, тогда как изменения вносятся "под капотом" для улучшения ситуации.

Где его использовать

Указанный шаблон очень хорошо работает там, где слишком трудно разобрать на части опорную схему. Помещая явную "обертку" вокруг схемы и давая понять, что

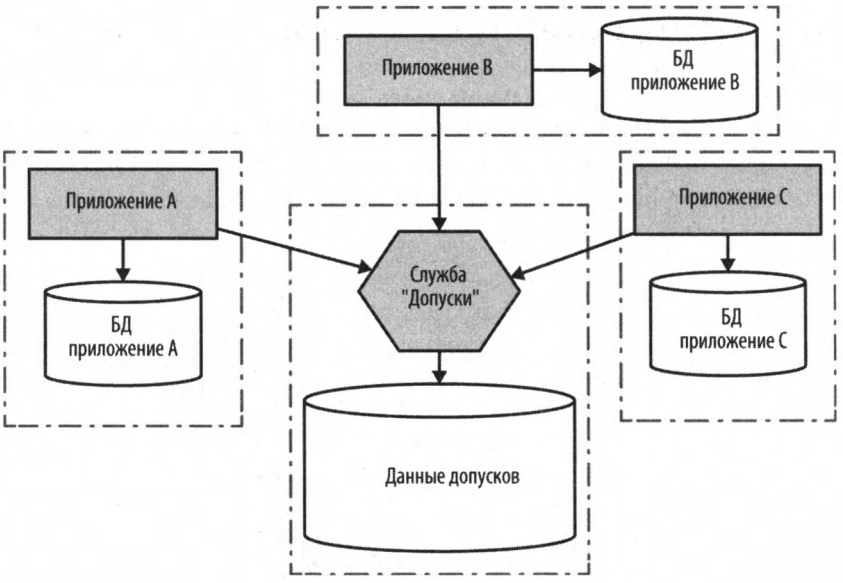


Рис. 4.6. Уменьшение зависимости от центральной базы данных с помощью шаблона "Служба обертывания базы данных"

данные будут доступны только через эту схему, вы, по меньшей мере, затормаживаете дальнейшее развитие базы данных. Указанный шаблон четко определяет, что является "вашим", по сравнению с тем, что является "чужим". Думаю, что этот подход также работает лучше всего, когда вы выравниваете владение как базовой схемой, так и слоем службы по одной и той же группе. API службы должен надлежаще восприниматься как управляемый интерфейс с соответствующим надзором за тем, как изменяется слой API. Подобный подход также выгоден для вышестоящих приложений, поскольку они легче "понимают", как они используют нижестоящую схему. Он делает такие мероприятия, как установка заглушек для целей тестирования, гораздо более управляемыми.

Указанный шаблон имеет преимущества перед использованием простой проекции базы данных. Прежде всего, вы не ограничены предоставлением проекции, отображаемой в структуры существующих таблиц, вы можете написать код в своей службе обертывания, который будет предоставлять гораздо более изощренные проекции на опорные данные. Служба обертывания также принимает операции записи (через вызовы API). Разумеется, внедрение этого шаблона требует от вышестоящих потребителей внесения изменений, им придется перейти от прямого доступа к БД к вызовам API.

В идеале, применение этого шаблона будет отправной точкой для более фундаментальных изменений, давая вам время на то, чтобы разложить схему под вашим слоем API. Можно утверждать, что мы просто накладываем "повязку" на проблему, вместо того чтобы обратиться к решению первичного затруднения. Тем не менее, в духе поступательного улучшения, я думаю, что указанный шаблон имеет много возможностей для этого.

Шаблон: "Интерфейс база-данных-как-служба"

Иногда клиентам нужна база данных только для запросов. Это бывает обусловлено тем, что им нужно запрашивать или извлекать большие объемы данных, или, возможно, потому, что внешние стороны уже используют цепочки инструментов, для работы с которыми требуется конечная точка на базе SQL (подумайте о таких инструментах, как Tableau, которые часто применяются для получения информации о бизнес-метриках). В этих ситуациях имеет смысл давать клиентам возможность просматривать данные, которыми ваша служба управляет в базе данных, но мы должны позаботиться об отделении базы данных, которую мы выставляем наружу, от базы данных, которую мы используем внутри контура нашей службы.

Один из подходов, который на моем опыте работает хорошо, состоит в создании выделенной базы данных, предназначенной для выставления наружу в качестве конечной точки с доступом только для чтения, и заполнении этой базы данных, когда данные в опорной базе данных изменяются. Практически, служба выставляет базу данных наружу внешним потребителям, точно так же, как она выставляет наружу поток событий в качестве единой конечной точки, и синхронный API — в качестве другой конечной точки. На рис. 4.7 мы видим пример службы "Заказ", которая выставляет наружу конечную точку с чтением/записью через API и базу данных в качестве интерфейса с доступом только для чтения. Механизм отображения берет изменения во внутренней базе данных и выясняет, какие изменения необходимо внести во внешнюю базу данных.

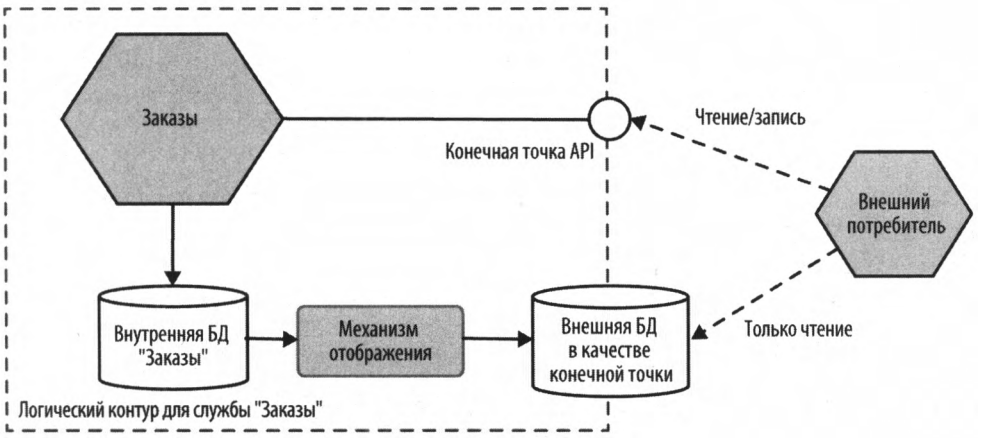


Рис. 4.7. Выставление выделенной базы данных в качестве конечной точки позволяет внутренней базе данных оставаться скрытой

Шаблон базы отчетных данных

Мартин Фаулер уже задокументировал этот подход в шаблоне базы отчетных данных (reporting database)[6], тогда почему я здесь применил другое название? Чем больше людей я опрашивал, тем больше я понимал, что, хотя распространенным применени-

[6] См. **http://bit.ly/2kWW9Ir**.

ем данного шаблона является отчетность, это не единственная причина, по которой люди используют указанный метод. Возможность разрешать клиентам определять нерегламентированные запросы имеет более широкую область применения, чем традиционные пакетные рабочие потоки. Поэтому, хотя указанный шаблон, вероятно, чаще всего ориентирован на поддержку отчетности, мне требовалось другое название, которое отражало бы тот факт, что он имеет более широкую применимость.

Механизм отображения может игнорировать изменения полностью, выставлять наружу изменения напрямую или делать что-то между ними. Ключевым моментом является то, что механизм отображения выступает в качестве слоя абстракции между внутренней и внешней базами данных. Когда наша внутренняя база данных изменяет структуру, механизму отображения нужно будет измениться для обеспечения согласованности базы данных, обращенной в публичное пространство. Практически во всех случаях механизм отображения будет отставать от операций записи, выполняемых во внутреннюю базу данных. В типичной ситуации это отставание определяется вариантом имплементации механизма отображения. Следовательно, клиенты, считывающие данные из выставленной наружу базы данных, должны понимать, что они видят потенциально устаревшие данные, и вы можете найти целесообразным программно выставлять информацию о том, когда внешняя база данных была обновлена в последний раз.

Имплементация механизма отображения

Тонкость здесь кроется в выработке ответа на вопрос, как обновлять, а именно, как имплементировать механизм отображения. Мы уже рассмотрели систему захвата изменений в данных, которая здесь была бы отличным вариантом выбора. На самом деле, она, вероятно, будет самым надежным решением, а также обеспечит наиболее актуальную проекцию. Еще один вариант — простое копирование данных пакетным процессом, хотя это бывает проблематичным, поскольку приводит к более длительному отставанию между внутренней и внешней базами данных, и с некоторыми схемами трудно определять то, какие данные должны быть скопированы. Третьим вариантом может быть прослушивание событий, запущенных из затрагиваемой службы, и использование его для обновления внешней базы данных.

В прошлом для улаживания этого вопроса я бы использовал пакетное задание. Сегодня же я бы, вероятно, задействовал выделенную систему захвата изменений в данных, возможно, что-то вроде Debezium[7]. Я слишком часто попадал в ситуации, когда пакетные процессы не работали или же работали слишком долго. Поскольку мир уходит от пакетных заданий и хочет получать данные быстрее, пакетная обработка уступает место реальному времени. Установка системы захвата изменений в данных на свое место для улаживания этого вопроса имеет смысл, в особенности, если вы подумываете о ее применении для выставления событий наружу за пределы контура службы.

[7] См. **https://github.com/debezium/debezium**.

Сравнение с проекциями базы данных

Этот шаблон выглядит более изощренным, чем простая проекция базы данных. Проекции базы данных обычно привязаны к тому или иному стеку технологий: если я хочу представить проекцию базы данных Oracle, то как лежащая в основе база данных, так и схема, в которой размещаются проекции, работают в Oracle. При таком подходе база данных, которую мы выставляем наружу, может иметь совершенно другой стек технологий. Внутри нашей службы можно было бы задействовать Cassandra, но представлять традиционную базу данных на базе SQL в качестве конечной точки, обращенной в публичное пространство.

Этот шаблон обеспечивает более высокую гибкость, чем проекции базы данных, но с добавленной стоимостью. Если потребности ваших потребителей могут быть удовлетворены с помощью простой проекции базы данных, то в первую очередь имплементация, скорее всего, обойдется в меньший объем работы. Просто учтите, что шаблон накладывает ограничения на то, как этот интерфейс будет эволюционировать. Вы можете начать с использования проекции базы данных и подумать о смещении в сторону выделенной базы отчетных данных в дальнейшем.

Где его использовать

Очевидно, что, поскольку база данных, выставляемая наружу в качестве конечной точки, доступна только для чтения, указанный шаблон полезен лишь для клиентов, которым требуется доступ только для чтения. Он очень хорошо укладывается в варианты использования, связанные с отчетностью, — ситуации, когда вашим клиентам требуется соединять большие объемы данных, хранящихся в той или иной службе. Эта идея может быть расширена за счет дальнейшего импортирования данных этой базы данных в более крупное хранилище данных, обеспечивая возможность опрашивать данные из нескольких служб. Я рассказываю об этом подробнее в *главе 5* книги "Создание микросервисов".

Не стоит недооценивать работу, необходимую для обеспечения надлежащей актуальности проекции внешней базы данных. В зависимости от того как имплементирована ваша текущая служба, это мероприятие может стать сложным.

Передача владения

До сих пор мы на самом деле не решали проблему, которая лежит в основании. Мы просто накладывали самые разные "повязки" на большую совместную базу данных. Прежде чем мы начнем думать о запутанной работе по вытаскиванию данных из гигантской монолитной базы данных, нам нужно подумать о том, где затрагиваемые данные фактически располагаются. Когда вы выделяете службы из монолита, некоторые данные должны уходить вместе с вами, а некоторые из них должны оставаться там, где они есть.

Если мы внедряем идею, в которой микрослужбы инкапсулируют логику, ассоциированную с одним или несколькими агрегатами, то нам также необходимо перене-

сти управление их внутренним состоянием и ассоциированными данными в собственную схему микрослужбы. С другой стороны, если нашей новой микрослужбе требуется взаимодействовать с агрегатом, который по-прежнему находится во владении монолита, то мы должны выставить эту возможность наружу через четко определенный интерфейс. Давайте теперь рассмотрим эти два варианта.

Шаблон: "Монолит с выставлением агрегата наружу"

На рис. 4.8 показано, что нашей новой службе "Выписка счетов-фактур" нужно обращаться к разнообразной информации, которая не имеет прямого отношения к управлению выпиской счетов. По крайней мере, ей нужна информация по нашим текущим "Сотрудникам" в целях управления трудовыми потоками по утверждению счетов-фактур. В настоящее время все эти данные находятся в базе данных монолита. Выставляя информацию о наших "Сотрудниках" наружу через конечную точку службы (это может быть API или поток событий) на самом монолите, мы четко указываем, какая информация требуется службе "Счет-фактура".

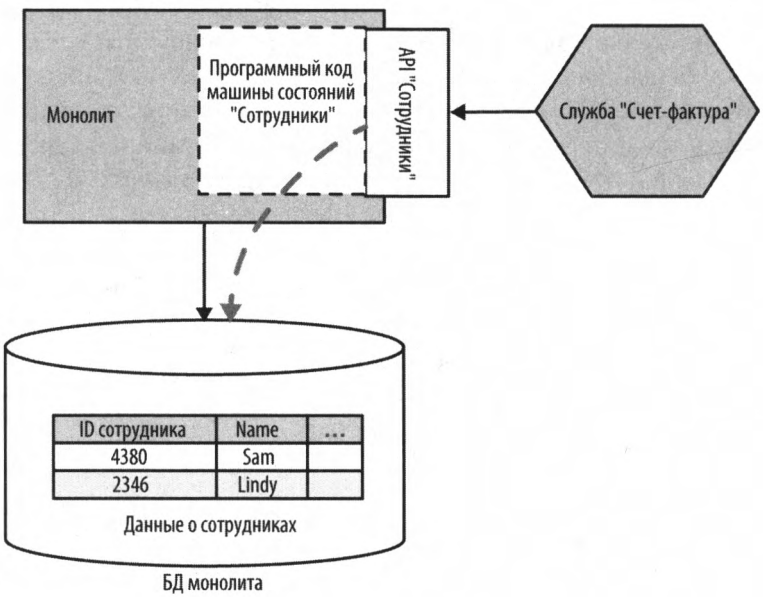


Рис. 4.8. Выставление информации из монолита наружу через надлежащий интерфейс микрослужбы позволяет нашей новой микрослужбе обращаться к этой информации

Мы хотим думать о наших микрослужбах как о комбинациях поведения и состояния; я уже обсуждал идею о микрослужбах как содержащих одну или несколько машин состояний, которые управляют доменными агрегатами. При выставлении агрегата наружу из монолита, мы хотим рассуждать в тех же понятиях. Монолит по-прежнему "владеет" понятием того, что является и не является допустимым

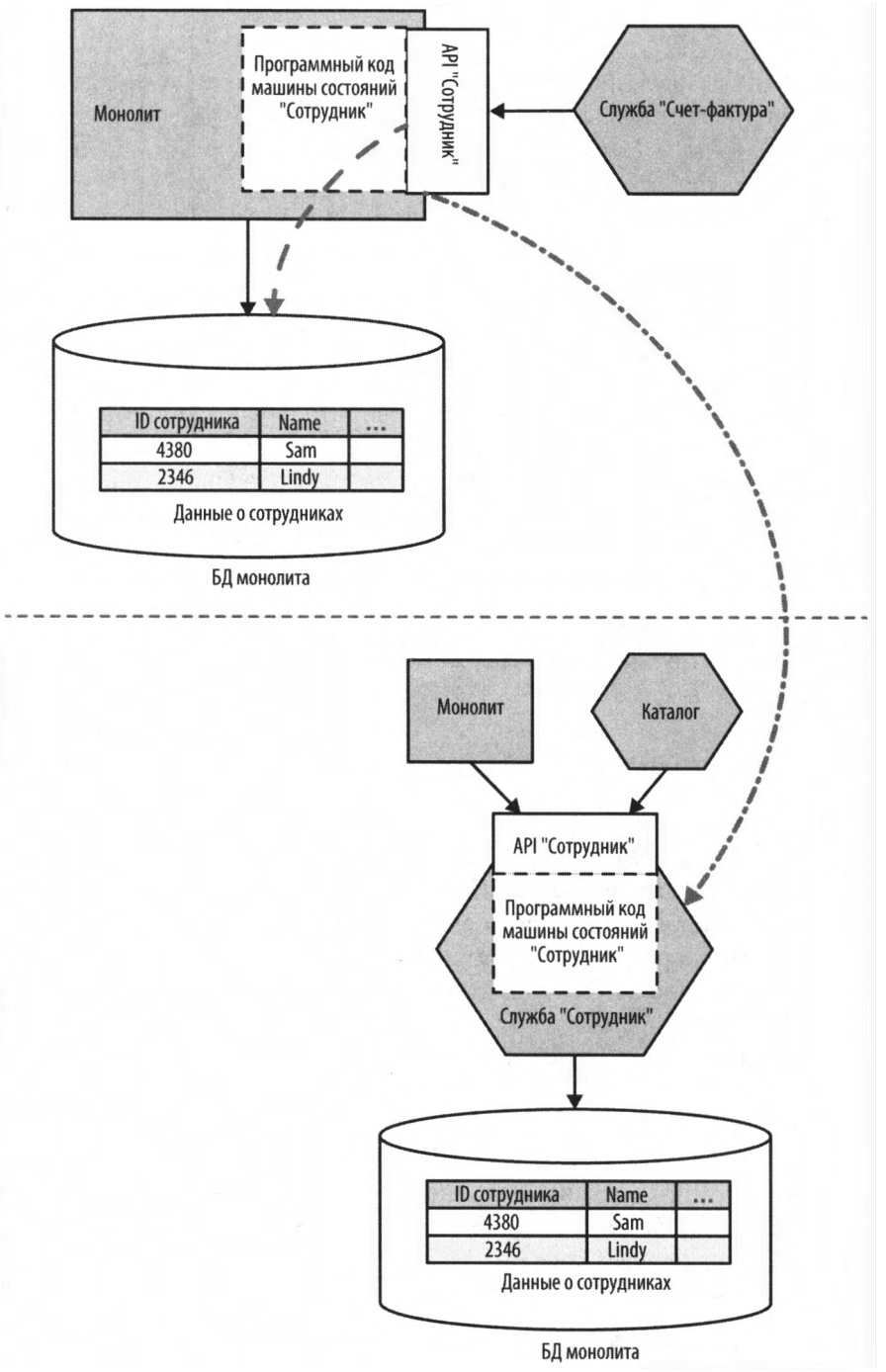


Рис. 4.9. Использование объема существующей конечной точки для приведения в действие извлечения новой службы "Сотрудник"

изменением состояния; мы не хотим трактовать это просто как "обертку" вокруг базы данных.

Помимо просто выставления данных наружу, мы выставляем операции, которые позволяют внешним сторонам запрашивать текущее состояние агрегата и запрашивать новые переходы из состояния в состояние. Мы по-прежнему можем решить ограничить то, какое состояние агрегата выставляется изнутри контура нашей службы наружу, и ограничить то, какие операции перехода из состояния в состояние могут запрашиваться снаружи.

В качестве пути к большему числу служб

Определяя потребности службы "Выписка счетов-фактур" и в явной форме выставляя наружу информацию, необходимую в четко определенном интерфейсе, мы встаем на путь потенциального обнаружения контуров будущих служб. В этом примере очевидный шаг — далее извлечь службу "Сотрудники", как мы видим на рис. 4.9. Выставив наружу API для данных, связанных с сотрудниками, мы уже прошли долгий путь к пониманию того, каковы потребности пользователей новой службы "Сотрудники".

Разумеется, если мы извлекаем сотрудников из монолита и монолит нуждается в данных об этих сотрудниках, то его, возможно, потребуется изменить так, чтобы он использовал новую службу!

Где его использовать

Когда данные, к которым вы хотите обратиться, по-прежнему находятся "во владении" базы данных, указанный шаблон хорошо работает, предоставляя вашим новым службам необходимый доступ. При извлечении служб необходимость в том, чтобы новая служба обращалась обратно в монолит для получения доступа к необходимым данным, скорее всего, будет оборачиваться немного большим объемом работы, чем прямой доступ к монолитной базе данных — но в долгосрочной перспективе эта идея гораздо лучше. Я бы подумал об использовании проекции базы данных вместо этого подхода только в том случае, если затрагиваемый монолит нельзя изменить с целью выставить наружу новые конечные точки. В таких случаях смогла бы работать проекция, взятая на базе данных монолита, как и ранее обсуждавшийся шаблон "Захват изменений в данных" (см. *раздел "Шаблон: захват изменений в данных"*), или создание шаблона выделенной службы обертывания базы данных (см. *раздел "Шаблон: служба обертывания базы данных"*) поверх схемы монолита, выставляя наружу информацию о "Сотруднике", которую мы хотим.

Шаблон: "Смена владельца данных"

Мы рассмотрели, что происходит, когда нашей новой службе "Счет-фактура" требуется обратиться к данным, которые принадлежат другой функциональности, как в предыдущем разделе, где нам нужно было обратиться к данным "Сотрудник".

Однако, что произойдет, когда мы рассматриваем данные, находящиеся в настоящее время в монолите, которые должны быть под контролем нашей только что извлеченной службы?

На рис. 4.10 мы описываем изменение, которое должно произойти. Нужно перенести наши счет-фактурные данные из монолита в новую "Счет-фактуру", поскольку как раз там находится контроль над жизненным циклом данных. Затем нам нужно изменить монолит так, чтобы трактовать службу "Счет-фактура" как источник истины для счет-фактурных данных, и изменить его таким образом, чтобы он вызывал конечную точку службы "Счет-фактура" для чтения данных или выполнения запроса об изменениях.

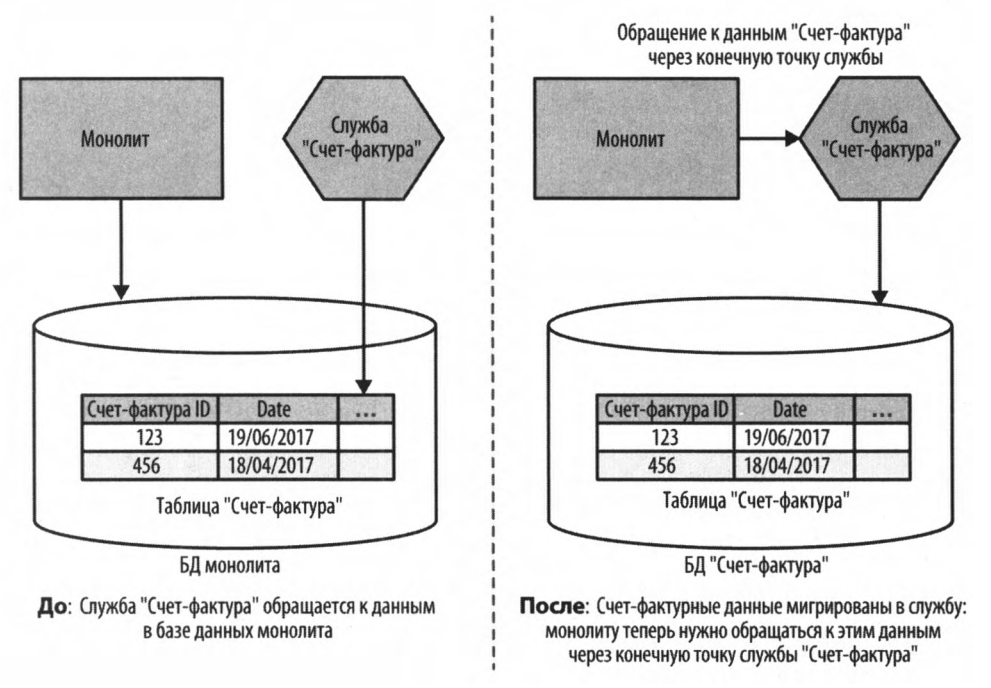


Рис. 4.10. Наша новая служба "Счет-фактура" берет на себя владение соответствующими данными

Бывает, что распутывание счет-фактурных данных существующей монолитной базы данных превращается в сложную проблему. Нам, возможно, придется рассмотреть влияние нарушения ограничений внешнего ключа, нарушения границ транзакций и многое другое — ко всем этим темам мы вернемся позже в этой главе. Если монолит будет изменен таким образом, что ему потребуется доступ к счет-фактурным данным лишь для чтения, то можно подумать о том, чтобы взять проекцию из базы данных службы "Счет-фактура", как показано на рис. 4.11. Однако при таком способе будут присутствовать все ограничения проекций базы данных. Гораздо предпочтительнее внести изменения в монолит так, чтобы он совершал вызовы новой службы "Счет-фактура" напрямую.

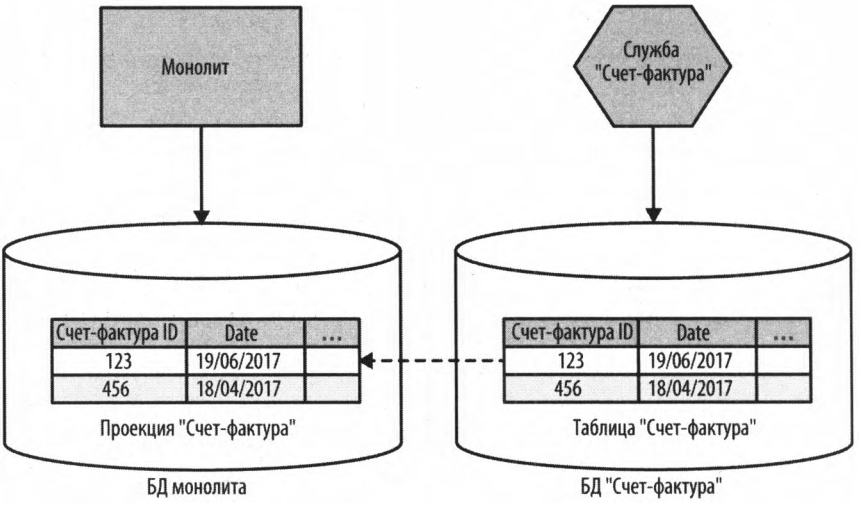


Рис. 4.11. Отображение счет-фактурных данных обратно в монолит в качестве проекции

Где его использовать

Указанный шаблон выглядит немного более четко очерченным. Если только что извлеченная служба инкапсулирует бизнес-логику и указанная логика изменяет некие данные, то эти данные должны находиться под контролем новой службы. Данные должны быть перенесены оттуда, где они находятся сейчас, в новую службу. Конечно, процесс перенесения данных из существующей базы данных далеко не простой. Собственно, на этом и будет сосредоточена остальная часть этой главы.

Синхронизация данных

Как мы обсуждали в *главе 3*, одна из выгод шаблона, подобного "Фикусу-удавке" состоит в том, что при переключении на новую службу мы затем можем переключиться обратно, если появляется затруднение. Проблема возникает, когда затрагиваемая служба управляет данными, которые должны поддерживаться в синхронизированном состоянии между монолитом и новой службой.

На рис. 4.12 мы видим такой пример. Мы находимся в процессе переключения на новую службу "Счет-фактура". Но новая служба и существующий эквивалентный программный код в монолите также управляют этими данными. Для поддержания возможности переключаться между имплементациями мы должны обеспечить, чтобы оба набора программного кода "видели" одни и те же данные и чтобы эти данные поддерживались согласованным образом.

Итак, какие тут у нас варианты? Ну, во-первых, нам нужно подумать о степени, в которой данные должны быть согласованы между двумя проекциями. Если какой-либо набор программного кода должен всегда видеть тотально согласованную проекцию счет-фактурных данных, то одним из наиболее простых подходов было бы обеспечить хранение данных в одном месте. Это, вероятно, приведет нас к тому,

что наша новая служба "Счет-фактура" будет считывать свои данные непосредственно из монолита в течение короткого промежутка времени, возможно, используя проекцию, как мы разведали в разделе "Шаблон: проекция базы данных". После успешного переключения мы можем мигрировать данные, как обсуждалось ранее в разделе "Шаблон: смена владельца данных". Однако опасения по поводу использования совместной базы данных невозможно преувеличить: вам следует ее рассматривать только как очень краткосрочную меру, как часть более полного извлечения. Если совместная база данных слишком долго остается на своем месте, то это приведет к значительной головной боли в долгосрочной перспективе.

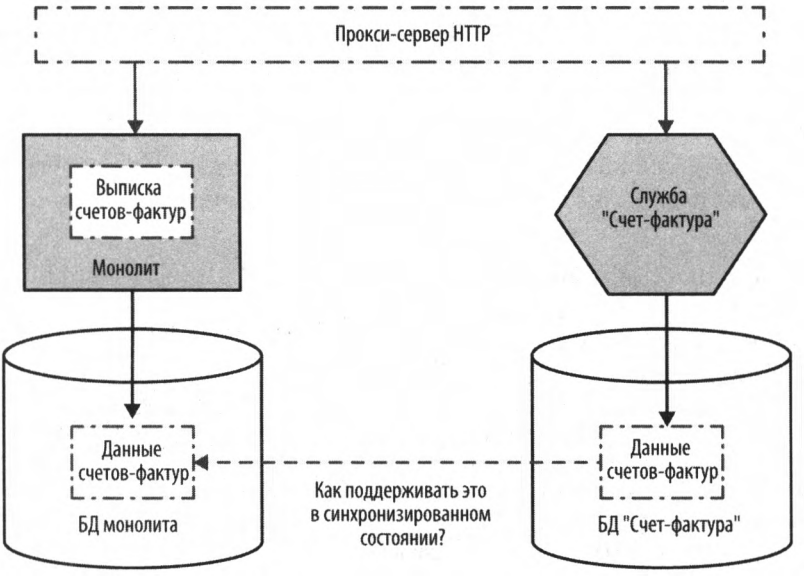


Рис. 4.12. Мы хотим использовать шаблон "Фикус-удавка" для мигрирования на новую службу "Счет-фактура", но эта служба управляет состоянием

Если бы мы выполняли переключение методом "Большого взрыва" (чего я бы старался избегать), мигрируя одновременно код приложения и данные, то могли бы использовать пакетный процесс для копирования данных перед переключением на новую микрослужбу. Как только счет-фатурные данные будут скопированы в нашу новую микрослужбу, она сможет начать обслуживать трафик. Однако, что произойдет, если нам нужно будет вернуться к использованию функциональности существующей монолитной системы? Данные, измененные в схеме микрослужб, не будут отражены в состоянии монолитной базы данных, поэтому мы в итоге потеряем внутреннее состояние.

Еще один подход — подумать о поддержании двух баз данных в синхронизированном состоянии с помощью нашего кода. В этом случае операции записи в обе базы данных у нас будет выполнять либо монолит, либо новая служба "Счет-фактура". Это решение требует тщательного обдумывания.

Шаблон: "Синхронизировать данные в приложении"

Переключать данные с одного места на другое — мероприятие сложное в самые лучшие времена, но оно тем больше чревато последствиями, чем более ценными являются данные. Когда вы начинаете думать о присмотре за медицинской документацией, тщательное обдумывание того, как вы мигрируете данные, становится еще важнее.

Несколько лет назад консалтинговая компания Trifork участвовала в проекте, призванном помочь в хранении консолидированного представления медицинской документации датских граждан[8]. Первоначальная версия этой системы хранила данные в базе данных MySQL, но со временем стало ясно, что она, возможно, не подойдет для задач, с которыми система столкнется. Было принято решение использовать альтернативную базу данных, Riak. Надежда состояла в том, что Riak даст системе возможность лучше масштабироваться, для того чтобы справляться с ожидаемой нагрузкой, но также предложит улучшенные характеристики отказоустойчивости.

Существующая система хранила данные в одной базе данных, но при этом существовали лимиты на то, как долго система может находиться в оффлайновом режиме, и было чрезвычайно важно, чтобы данные не были потеряны. Поэтому требовалось решение, которое позволило бы компании не только перенести данные в новую базу данных, но и построить механизмы, которые верифицировали бы миграцию, а также в рабочем порядке имели быстрые механизмы отката.

Было принято решение, что приложение само будет выполнять синхронизацию между двумя источниками данных. Идея заключалась в том, что изначально существующая база данных MySQL останется источником истины, но в течение определенного периода времени приложение будет обеспечивать синхронизацию данных в MySQL и Riak. Через некоторое время Riak станет источником истины для приложения, и только после этого MySQL будет выведена из эксплуатации. Давайте рассмотрим этот процесс немного подробнее.

Шаг 1: массово синхронизировать данные

Первый шаг — добраться до точки, где у вас есть копия данных в новой базе данных. Для проекта медицинской документации это включало выполнение пакетной миграции данных из старой системы в новую базу данных Riak. Пока продолжался пакетный импорт, существующая система продолжала работать, поэтому источником данных для импорта был снимок данных, взятый из существующей системы MySQL (рис. 4.13). Это вызывает трудность в том плане, что после завершения пакетного импорта данные в исходной системе вполне могли измениться. В этом

[8] Подробную презентацию по этой теме можно увидеть в записи выступления Крестена Краба Торупа (Kresten Krab Thorup) "Riak на таблетках (и наоборот)" (Riak on Drugs (and the Other Way Around), **http://bit.ly/2m1CvLP**).

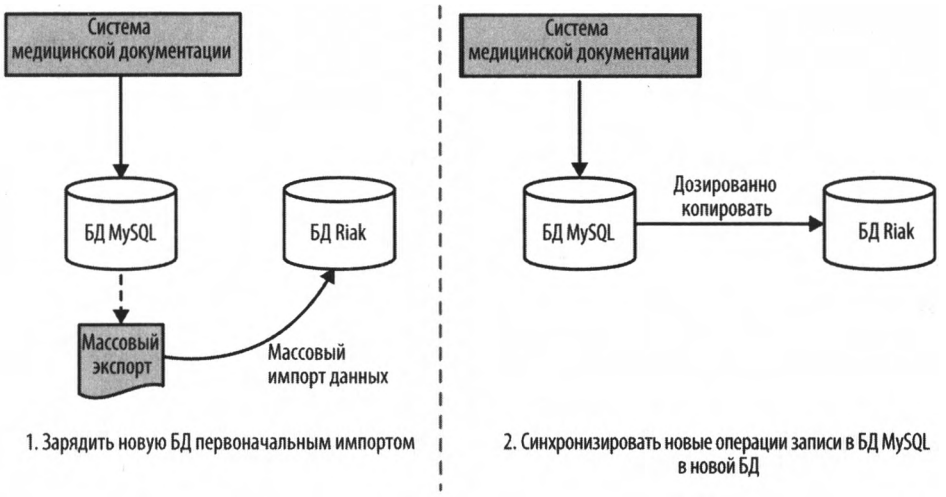


Рис. 4.13. Подготовка нового хранилища данных для синхронизации на основе приложения

случае, однако, было нецелесообразно переводить исходную систему в оффлайновый режим.

После завершения пакетного импорта был имплементирован процесс захвата изменений в данных, с помощью которого были применены изменения, внесенные с момента начала импорта. Это позволило привести Riak в синхронизированное состояние. После того как это было достигнуто, настало время развернуть новую версию приложения.

Шаг 2: синхронизировать при записи, читать из старой схемы

Теперь, когда обе базы данных были синхронизированы, мы развернули новую версию приложения, которая будет записывать все данные в обе базы данных, как показано на рис. 4.14. Цель на данном этапе — обеспечить, чтобы приложение пра-

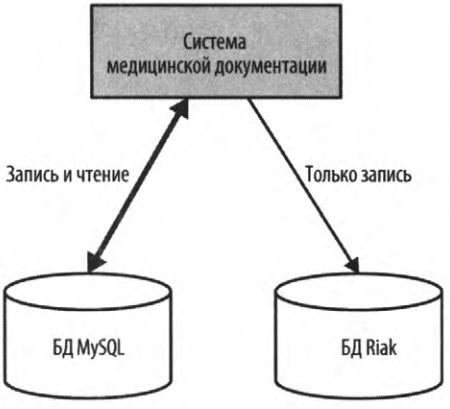


Рис. 4.14. Приложение поддерживает обе базы данных в синхронизированном состоянии, но одну использует только для чтения

вильно писало в оба источника и чтобы Riak вела себя в пределах приемлемых допусков. Тот факт, что все данные по-прежнему читались из MySQL, обеспечивал возможность извлечения данных из существующей базы данных MySQL даже в случае, если бы Riak "рухнула".

Только после того, как укрепилась достаточная уверенность в новой системе Riak, они перешли к следующему шагу.

Шаг 3: синхронизировать при записи, читать из новой схемы

На этом этапе было верифицировано, что чтение в Riak осуществляется нормально. Последний шаг — убедиться, что запись тоже работает. Простое изменение в приложении — и теперь Riak становится источником истины, как мы видим на рис. 4.15. Обратите внимание, что мы по-прежнему пишем в обе базы данных, поэтому при возникновении каких-либо затруднений у вас есть возможность откатить назад.

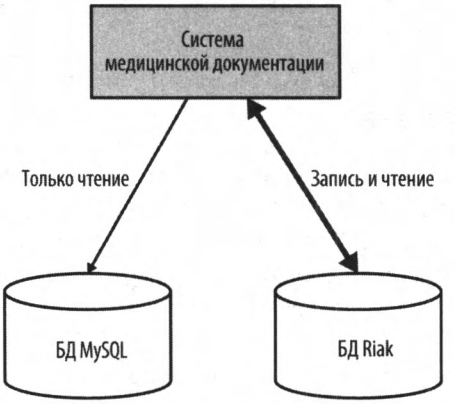


Рис. 4.15. Новая база данных теперь является источником истины, но старая база данных по-прежнему поддерживается в синхронизированном состоянии

После того как новая система будет в достаточной мере отлажена, старую схему можно безопасно удалить.

Зачем использовать этот шаблон

В случае с датской системой медицинской документации мы занимались одним приложением. Но мы говорим о ситуациях, когда хотим извлекать микрослужбы. Тогда действительно ли этот шаблон помогает? Первым делом следует учесть то, что этот шаблон имеет большой смысл, если вы хотите разложить схему *перед* выделением кода приложения. На рис. 4.16 мы видим именно такую ситуацию, когда сначала мы дублируем счет-фактурные данные.

При правильной имплементации оба источника данных всегда должны находиться в синхронизованном состоянии, что дает нам значительные выгоды в ситуациях,

когда требуется быстрое переключение между источниками для сценариев отката и т. д. Использование этого шаблона на примере датской системы медицинской документации представляется разумным из-за невозможности перевести приложение в оффлайновый режим в течение любого периода времени.

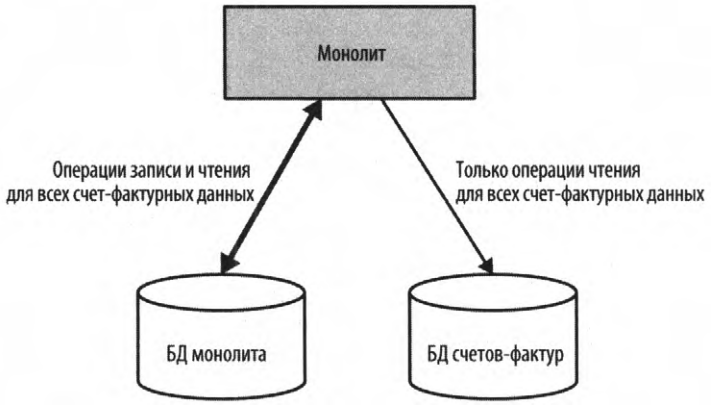


Рис. 4.16. Пример монолита, поддерживающего две схемы в синхронизованном состоянии

Где его использовать

Теперь, когда у вас есть как монолит, так и микрослужба, которая обращается к данным, вы можете подумать об использовании этого шаблона, но это чрезвычайно усложняется. На рис. 4.17 мы имеем такую ситуацию. Для того чтобы этот шаблон работал как надо, необходимо, чтобы и монолит, и микрослужба обеспечивали надлежащую синхронизацию во всех базах данных. Если одно из них ошибется, то у вас будут неприятности. Эта сложность значительно смягчается, если вы уверены в том, что в любой момент времени либо служба "Счет-фактура" выполняет операции записи, либо это делает счет-фактурная функциональность монолита.

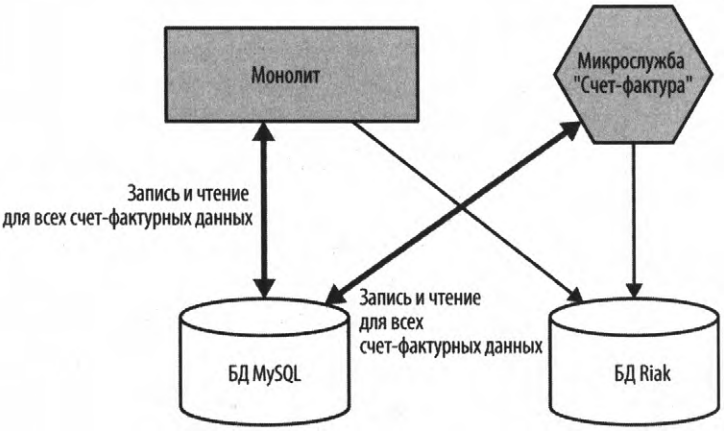


Рис. 4.17. Пример, когда и монолит, и микрослужба пытаются поддерживать две одинаковые схемы в синхронизованном состоянии

Это будет хорошо работать, если применить простой метод переключения, как мы обсуждали на примере шаблона "Фикус-удавка". Если, однако, запросы могут затрагивать либо монолитную счет-фактурную функциональность *либо* новую счет-фактурную функциональность, возможно, в рамках "канареечного" релиза, то вы, возможно, не захотите использовать этот шаблон, т. к. результирующая синхронизация будет запутанной.

Шаблон: "Трассировочная запись"

Шаблон "Трассировочная запись" (tracer write), очерченный на рис. 4.18, можно назвать вариацией шаблона "Синхронизовать данные в приложении" (см. *раздел "Шаблон: синхронизовать данные в приложении"*). С помощью трассировочной записи мы переносим источник истины для данных в поступательном режиме, допуская во время миграции наличие двух источников истины. Вы определяете новую службу, которая будет содержать перенесенные данные. Текущая система попрежнему ведет регистрацию этих данных локально, но во время внесения изменений также обеспечивает, чтобы эти данные записывались в новую службу через ее интерфейс. Существующий код можно изменить так, чтобы начать обращаться к новой службе, и как только вся функциональность будет использовать новую службу в качестве источника истины, старый источник истины можно вывести из эксплуатации. Следует уделять тщательное внимание тому, как данные синхронизируются между двумя источниками истины.

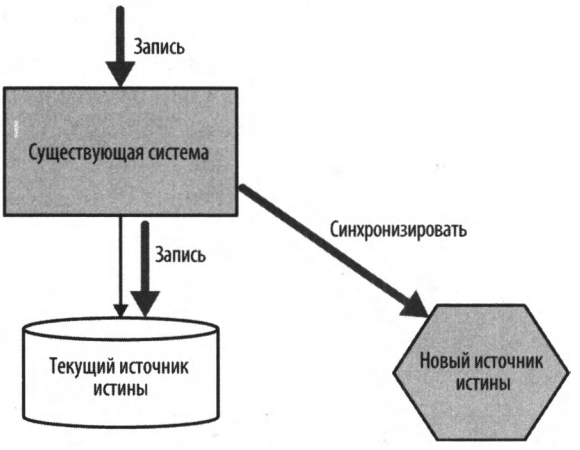


Рис. 4.18. Трассировочная запись позволяет осуществлять поступательную миграцию данных из одной системы в другую, обустраивая два источника истины во время миграции

Желание иметь единственный источник истины является совершенно рациональным. Это позволяет нам обеспечить согласованность данных, контролировать доступ к этим данным и сокращать расходы на сопровождение. Проблема заключается в том, что если мы настаиваем на единственном источнике истины для части данных, то вынуждены оказаться в ситуации, когда изменение места расположения этих данных становится одним большим переключением. До релиза источником

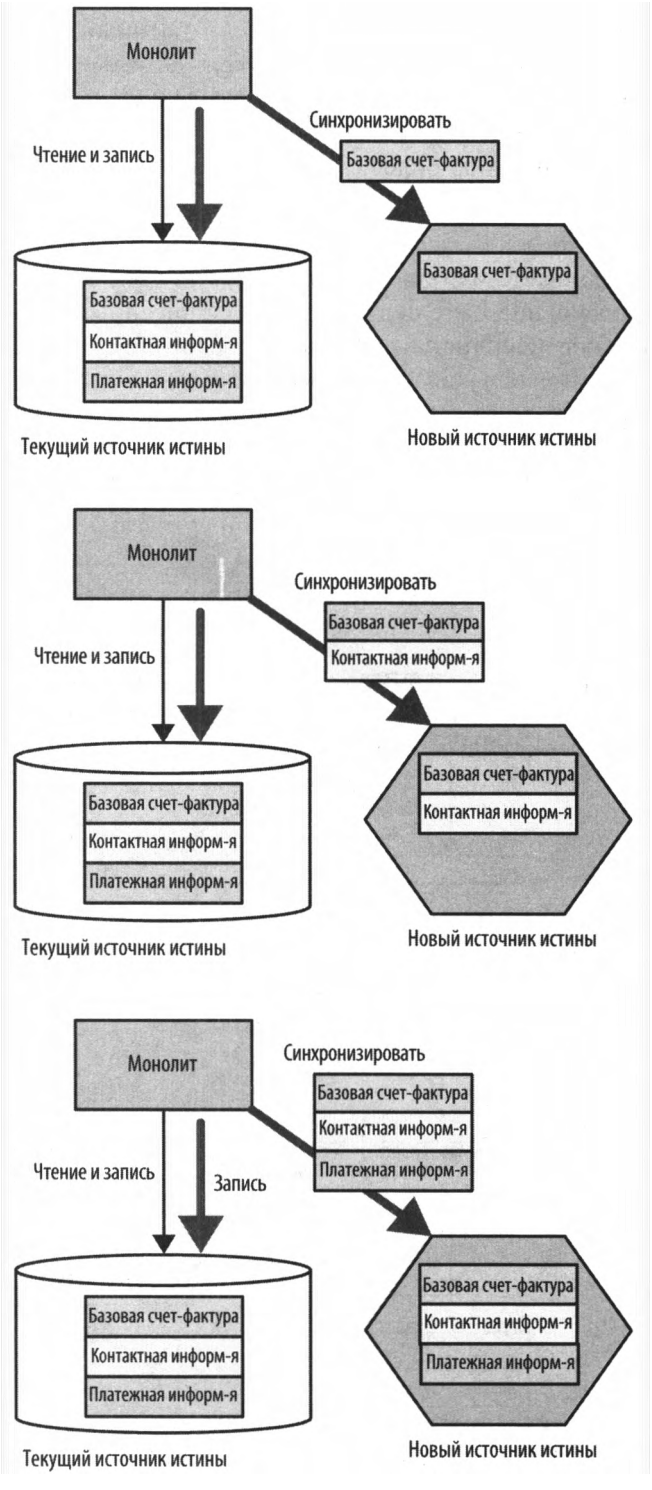


Рис. 4.19. Поступательный перенос счет-фактурной информации из монолита в нашу службу "Счет-фактура"

истины служит монолит. После релиза источник истины — наша новая микрослужба. Трудность в том, что во время этого обмена самые разные вещи могут пойти не так. Шаблон, подобный "Трассировочной записи", позволяет осуществлять фазированное переключение, уменьшая влияние каждого релиза в обмен на более терпимое отношение к наличию более одного источника истины.

Причина, почему этот шаблон называется трассировочной записью, заключается в том, что вы начинаете с небольшого набора синхронизируемых данных и с течением времени увеличиваете их объем, одновременно увеличивая число потребителей нового источника данных. Если взять пример, описанный на рис. 4.12, где счет-фактурные данные переносились из монолита в нашу новую микрослужбу "Счет-фактура", то мы могли бы сначала синхронизировать базовые счет-фактурные данные, затем мигрировать контактную информацию счета-фактуры и, наконец, синхронизировать платежные данные, как показано на рис. 4.19.

Другие службы, которые хотели бы получать счет-фактурную информацию, будут выбирать источник этой информации либо из монолита, либо из самой новой службы, в зависимости от того, какая информация им нужна. Если им по-прежнему требуется информация, имеющаяся только в монолите, то им придется подождать до тех пор, пока эти данные и вспомогательная функциональность не будут перенесе-

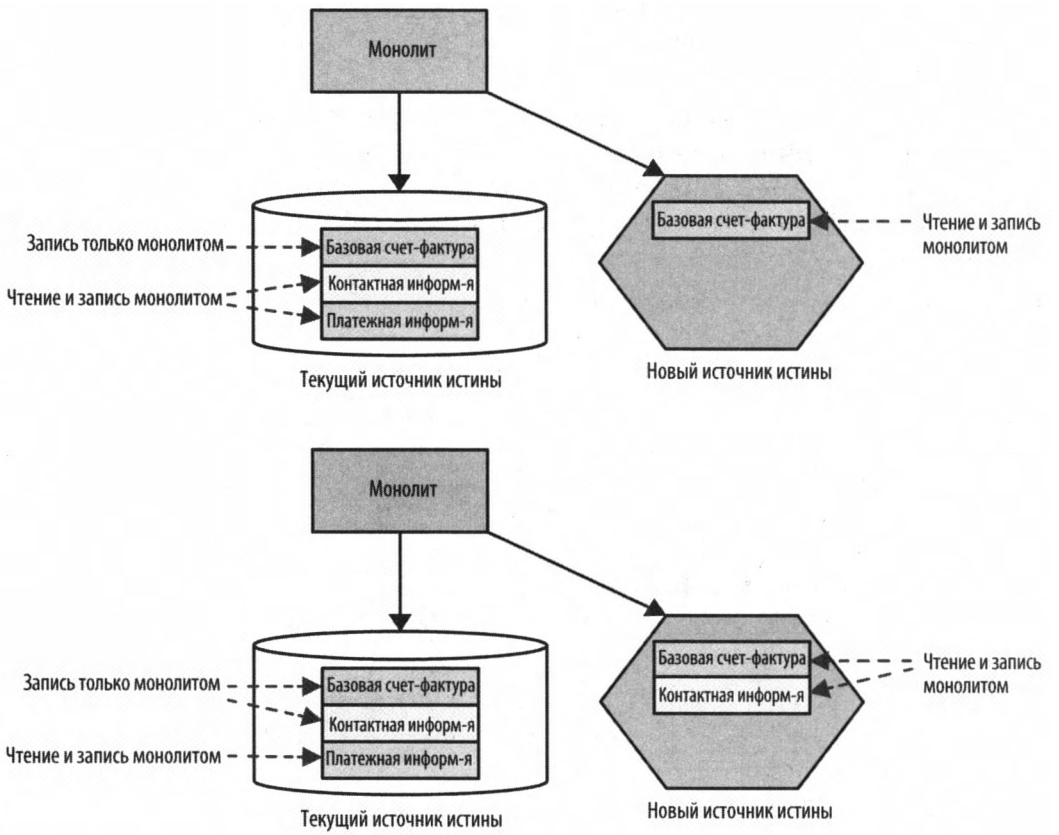


Рис. 4.20. Вывод из эксплуатации старого источника истины с помощью трассировочной записи

ны. Как только данные и функциональность будут иметься в новой микрослужбе, потребители смогут переключиться на новый источник истины.

Цель нашего примера — мигрировать всех потребителей на использование службы "Счет-фактура", включая сам монолит. На рис. 4.20 мы видим пример двух этапов миграции. Первоначально мы пишем только базовую счет-фактурную информацию для обоих источников истины. Как только мы подтвердим, что эта информация надлежаще синхронизирована, монолит может начать читать свои данные из новой службы. По мере того как синхронизируется все больше данных, монолит использует новую службу в качестве источника истины для все большего объема данных. Как только все данные синхронизированы и последний потребитель старого источника истины был переключен, мы можем прекратить синхронизацию данных.

Синхронизация данных

Самая большая задача, которую необходимо решить при использовании шаблона "Трассировочная запись", связана с проблемой, возникающей в любой ситуации, когда данные дублируются, — несогласованность. Для ее решения у вас есть несколько вариантов:

- *Писать в один источник* — все записи отправляются в один из источников истины. После осуществления записи данные синхронизируются с другим источником истины.
- *Отправлять записи в оба источника* — все запросы на запись со стороны вышестоящих клиентов отправляются в оба источника истины. Это происходит, убедившись, что клиент обращается к каждому источнику истины сам или полагаясь на посредника в том, что он протранслирует запрос в каждую нижестоящую службу.
- *"Высеивать" записи в любой из двух источников* — клиенты могут отправлять запросы на запись в любой источник истины, и за кулисами данные синхронизируются в двустороннем режиме между системами.

Два отдельных варианта отправки операций записи в оба источника истины или отправки в один источник истины и опора на некоторую форму фоновой синхронизации выглядят работоспособными решениями, и пример, который мы вскоре рассмотрим, использует оба этих метода. Однако, хотя эта ситуация технически возможна, ее — когда записи выполняются либо в один источник истины, либо в другой — следует избегать, поскольку она требует двусторонней синхронизации (что бывает очень трудно достичь).

Во всех этих случаях будет наблюдаться некоторая задержка в согласовании данных в обоих источниках истины. Продолжительность этого окна несогласованности будет зависеть от нескольких факторов. Например, если вы копируете обновления из одного источника в другой с помощью ночного пакетного процесса, то второй источник истины может содержать данные, которые устарели на 24 часа. Если вы постоянно передаете обновления потоком из одной системы в другую, исполь-

зуя систему захвата изменений в данных, то окна несогласованности могут измеряться в секундах или меньше.

Как бы долго ни длилось это окно несогласованности, такая синхронизация дает нам то, что называется конечной согласованностью[9] — в результате оба источника истины будут иметь одни и те же данные. Вам нужно будет понять, какой период несогласованности подходит в вашем случае, и учесть его при выборе способа имплементации синхронизации.

Важно, чтобы при поддержании такого рода двух источников истины у вас был какой-то процесс согласования для обеспечения работы синхронизации как задумано. Это может быть что-то простое, например, несколько SQL-запросов, которые вы выполняете к каждой базе данных. Но без проверки того, что синхронизация работает как задумано, вы можете столкнуться с несогласованностью между двумя системами и не понять этого, пока не станет слишком поздно. Очень разумно выполнять ваш новый источник истины в течение периода времени, когда у него нет потребителей, до тех пор пока вы не будете удовлетворены тем, как все работает — что, как мы разведаем в следующем разделе, как раз и сделали, например, в Square.

Пример: заказы в Square

Первоначально этим шаблоном со мной поделился Дерек Хаммер (Derek Hammer), разработчик из Square, и с тех пор в общей практике я нашел другие примеры его использования[10]. Дерек подробно описал его применение, помогая распутать часть домена компании Square, связанного с заказом доставки еды на вынос. В первоначальной системе концепция единого "Заказа" управляла несколькими трудовыми потоками: один для клиентов, заказывающих еду, другой для ресторана, готовящего еду, и третий управляемый трудовой поток состоял в том, что водители доставки забирали еду и отправляли ее клиентам. Потребности трех заинтересованных сторон различаются, и, хотя все эти стороны работают с одним и тем же "Заказом", значение этого "Заказа" для каждого из них различно. Клиенту важно то, что он выбрал для доставки, и то, за что ему нужно заплатить. Ресторану требуется знать, что нужно приготовить и укомплектовать. И какой заказ водителю нужно своевременно доставить из ресторана клиенту. Несмотря на эти разные потребности, код и ассоциированные с "Заказом" данные связаны между собой.

Наличие всех этих трудовых потоков, объединенных в единую концепцию "Заказа", было источником уже упоминавшейся "конкуренции за доставку" — разные разработчики, которые пытаются внести изменения, соответствующие разным вариантам

---

[9] Для справки: конечная согласованность (eventual consistency) — это модель согласованности, используемая в распределенных вычислениях для достижения высокой доступности, которая неофициально гарантирует, что если в отдельно взятый элемент данных не будет внесено никаких новых обновлений, то в конечном итоге все обращения к этому элементу вернут последнее обновленное значение — *Пер.*

[10] Сандита Ханда (Sangeeta Handa) поделилась тем, как Netflix использовал этот шаблон в рамках своих миграций данных на конференции QCon SF, и Даниэль Брайант (Daniel Bryant) впоследствии составил о нем хорошую тематическую подборку (**http://bit.ly/2m1EwHT**).

использования, будут мешать друг другу, поскольку всем им нужно вносить изменения в одну и ту же часть кодовой базы. Square хотела разложить "Заказ" так, чтобы изменения в каждом трудовом потоке могли делаться изолированно, а также активировать разные потребности в масштабировании и робастности.

Создание новой службы

Первым шагом было создание новой службы "Исполнения", как показано на рис. 4.21, которая будет управлять данными "Заказа", ассоциированными с рестораном и водителями доставки. Эта служба в будущем должна была стать новым источником истины для подмножества данных "Заказа". Первоначально эта служба просто выставляла наружу функциональность, позволяющую создавать сущности, связанные с "Исполнением". После того как новая служба была выпущена в "прямой эфир", у компании имелся фоновой работник, который копировал данные, связанные с "Исполнением", из существующей системы в новую службу "Исполнения". Этот фоновый работник просто использовал API, выставляемый наружу службой "Исполнения", вместо прямой вставки в базу данных, избегая необходимости прямого обращения к базе данных.

Рис. 4.21. Новая служба "Исполнения" использовалась для репликации данных, связанных с исполнением заказов, из существующей системы

Фоновый работник контролировался посредством флажка функции, который можно было включить или выключить для остановки процесса копирования. Это позволило легко отключать указанный процесс, если работник становился причиной каких-либо проблем в производстве. Они прогнали эту систему в производстве в течение достаточного объема времени, для того чтобы получить уверенность в том, что синхронизация работала правильно. Как только они были довольны тем, что фоновый работник работал так, как ожидалось, этот флажок функции был удален.

Синхронизировать данные

Одна из трудностей с такого рода синхронизацией заключается в том, что она является односторонней. Изменения, внесенные в существующую систему, привели к тому, что данные, связанные с исполнением заказов, записывались в новую службу "Исполнения" через ее API. Square решила эту проблему, обеспечив внесение всех обновлений в обеих системах, как показано на рис. 4.22. Однако не все обновления нужно было делать в обеих системах. Как объяснил Дерек, теперь, когда служба "Исполнения" представляла собой только подмножество концепции "Заказ", нужно было копировать только те изменения, внесенные в заказ, которые интересовали клиентов доставки или ресторана.

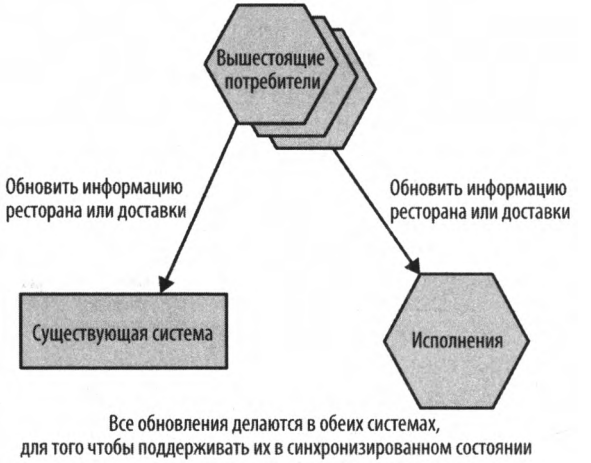


Рис. 4.22. Последующие обновления синхронизировались путем обеспечения того, чтобы все потребители делали надлежащие вызовы API в сторону обеих служб

Любой код, который изменял информацию, ориентированную на ресторан или доставку, следовало модифицировать так, чтобы делались два набора вызовов API: один — в сторону существующей системы, другой — в сторону той же микрослужбы. Эти вышестоящие клиенты также должны были улаживать любые условия возникновения ошибок, если запись в одну из них работала, а в другую — нет. Изменения в двух нижестоящих системах (существующая система заказов и новая служба "Исполнения") не делались атомарно. Это означает, что существовало короткое окно, в котором изменение видно в одной системе, но еще не видно в другой. До тех пор пока оба изменения не будут применены, вы видите несогласованность между двумя системами. Это одна из форм конечной согласованности, о которой мы говорили ранее.

С точки зрения возможной последовательной природы информации о "Заказе", это не было проблемой для указанного конкретного варианта использования. Данные синхронизировались между двумя системами достаточно быстро, что не влияло на пользователей системы.

Если бы для управления обновлениями "Заказов" вместо вызовов API в компании Square использовалась событийно-обусловленная система, то они могли бы поду-

мать об альтернативной имплементации. На рис. 4.23 представлен единый поток сообщений, который активирует изменения в состоянии "Заказа". И существующая система, и новая служба "Исполнения" получают одни и те же сообщения. Вышестоящим клиентам не нужно знать о том, что для этих сообщений существуют многочисленные клиенты; это то, что может быть урегулировано с помощью брокера в стиле "публикация-подписка".

Рис. 4.23. Альтернативный подход к синхронизации может заключаться в том, чтобы оба источника истины подписывались на одинаковые события

Донастройка архитектуры Square под событийную обусловленность только для удовлетворения такого рода варианта использования выльется в большую работу. Но если вы уже используете систему на основе событий, то вам будет проще управлять процессом синхронизации. Также стоит отметить, что такая архитектура все равно в итоге будет демонстрировать согласованное поведение, поскольку вы не можете гарантировать, что и существующая система, и служба "Исполнения" будут обрабатывать одно и то же событие в то же самое время.

Мигрирование потребителей

Теперь, когда новая служба "Исполнения" содержит всю необходимую информацию для трудовых потоков ресторана и водителя доставки, код, управлявший этими трудовыми потоками, может начать переключаться на использование новой службы. Во время этой миграции может привноситься дополнительная функциональность для поддержки потребностей этих потребителей. Первоначально служба "Исполнения" требовалась только для имплементации API, который позволял создавать новые записи для фонового работника. По мере миграции новых потребителей оцениваются их потребности, и в службу добавляется новая функциональность с целью их поддержки.

Эта поступательная миграция данных, а также изменение потребителей, чтобы они использовали новый источник истины, в случае Square оказались весьма эффектив-

ными. По словам Дерека, достижение той точки, где все потребители переключились, в значительной степени оказалось разочарованием. Это было просто еще одним крохотным изменением, сделанным во время рутинного релиза (еще одна причина, по которой в этой книге я так сильно ратую за поступательную миграцию!).

С точки зрения доменно-обусловленного дизайна вы, конечно, можете утверждать, что вся функциональность, связанная с водителями доставки, клиентами и ресторанами, представляла разные ограниченные контексты. С этой точки зрения Дерек предположил, что он в идеале мог бы подумать о дальнейшем разбиении этой службы "Исполнения" на две службы: одну — для ресторанов, а другую — для водителей доставки. Тем не менее, несмотря на наличие возможностей для дальнейшей декомпозиции, эта миграция оказалась весьма успешной.

В случае с Square продублированные данные было решено оставить. Благодаря тому, что информация о заказах, связанных с рестораном и доставкой, осталась в существующей системе, компания обеспечила видимость этой информации в случае недоступности службы "Исполнения". Разумеется, при этом синхронизация тоже должна остаться на своем месте. Интересно, будет ли это со временем подвергнуто ревизии. Как только появится достаточная уверенность в работоспособности службы "Исполнения", удаление фонового работника и необходимости в том, чтобы потребители делали два набора вызовов на обновление, вполне помогут оптимизировать указанную архитектуру.

Где его использовать

Синхронизация, вероятно, будет имплементироваться там, где находится бо́льшая часть работы. Если вы можете избежать потребности в двусторонней синхронизации и вместо этого использовать более простые варианты, которые были описаны здесь, то вы, вероятно, найдете, что этот шаблон имплементируется гораздо легче. Если вы уже пользуетесь событийно-обусловленной системой, или у вас имеется конвейер захвата изменений в данных, то в вашем распоряжении, вероятно, уже есть много строительных блоков, для того чтобы пустить в работу синхронизацию.

Необходимо тщательно продумать, как долго вы сможете допускать несогласованность между двумя системами. Некоторые варианты использования могут не волновать, другие потребуют, чтобы репликация была почти немедленной. Чем короче окно приемлемой несогласованности, тем сложнее будет имплементировать этот шаблон.

Разбиение базы данных

Мы уже подробно обсуждали проблемы использования баз данных в качестве метода интеграции многочисленных служб. Как уже должно быть совершенно ясно, я не поклонник этого! А значит, нам нужно найти стыки в базах данных тоже, для того чтобы аккуратно их выделить. Вместе с тем базы данных — звери хитрые. Прежде чем перейти к нескольким примерам подходов, мы должны кратко обсу-

дить вопрос о том, каким образом связаны логическое и физическое разделение баз данных.

Физическое и логическое разделение баз данных

Когда мы говорим о разложении наших баз данных в этом контексте, мы в первую очередь пытаемся достичь логического разделения, или сегментации. Одно ядро СУБД способно идеально разместить более одной логически разделенной схемы, как показано на рис. 4.24.

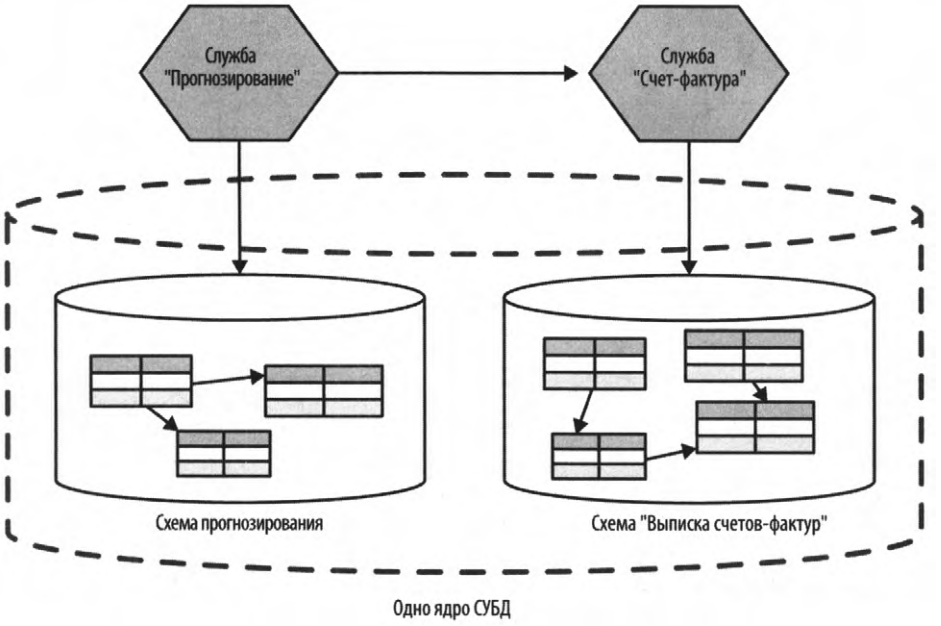


Рис. 4.24. Две службы с отдельными логическими схемами, работающими на одном физическом ядре СУБД

Мы могли бы пойти дальше и разместить каждую логическую схему также на отдельных ядрах СУБД, получая и физическое разделение тоже, как мы видим на рис. 4.25.

Почему возникает желание разложить свои схемы логически, но все-таки иметь их на одном ядре СУБД? Дело в том, что в своей основе логическая и физическая декомпозиция преследуют разные цели. Логическая декомпозиция обеспечивает более простое независимое изменение и сокрытие информации, в то время как физическая декомпозиция потенциально повышает робастность системы и помогает устранять конкуренцию за ресурсы, позволяя повышать пропускную способность или уменьшать задержку.

Когда мы разбиваем наши схемы баз данных логически, но сохраняем их на том же физическом ядре СУБД, как показано на рис. 4.24, мы имеем потенциальную единственную точку сбоя. Если ядро СУБД выходит из строя, то затрагиваются обе

службы. Однако мир не так прост. Многие ядра СУБД имеют механизмы, позволяющие избегать единственных точек сбоя, такие, как мультипримарные режимы баз данных, механизмы "теплой" отработки отказа и тому подобное. Собственно, в вашей организации, возможно, были приложены значительные усилия, для того чтобы создать высокоотказоустойчивый кластер базы данных, и будет трудно оправдать наличие нескольких кластеров из-за связанных с этим времени, усилий и стоимости (а досадные лицензионные сборы пойдут в нагрузку!).

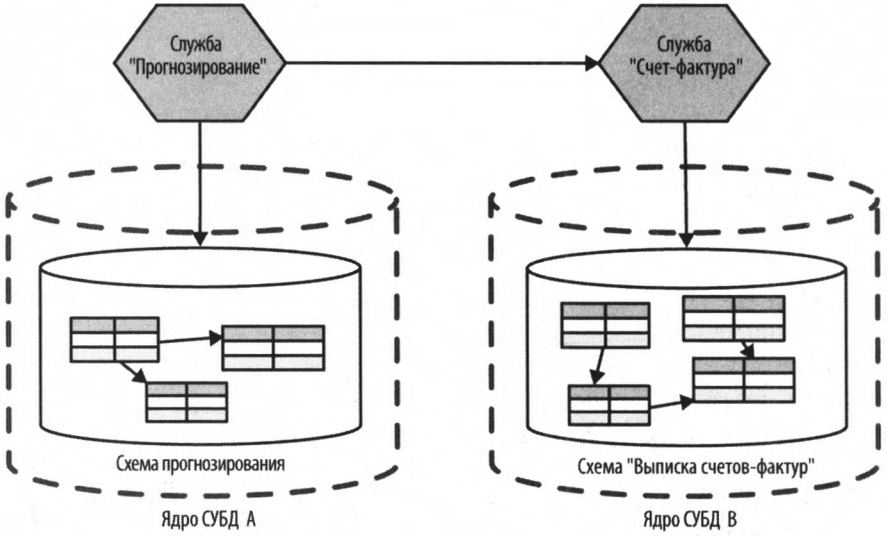


Рис. 4.25. Две службы, использующие отдельные логические схемы, каждая из которых работает на своем собственном физическом ядре СУБД

Еще одно соображение состоит в том, что, если вы захотите выставлять наружу проекции своей базы данных, то потребуется, чтобы многочисленные схемы совместно использовали одно и то же ядро СУБД. Возникнет необходимость расположить на одном и том же ядре СУБД как исходную базу данных, так и схемы, на которых размещаются проекции.

Конечно, для того чтобы у вас была даже возможность выполнять отдельные службы на разных физических ядрах СУБД, вам нужно уже разложить их схемы логически!

Что выделять сначала: базу данных или код?

До сих пор мы говорили о шаблонах, которые помогают работать с совместными базами данных и, надо надеяться, переходить к менее сопряженным моделям. В данный момент нам нужно осмотреть со всех сторон шаблоны декомпозиции базы данных. Однако, прежде чем мы это сделаем, нам необходимо обсудить вопрос установления последовательности. Извлечение микрослужбы не "делается" до тех пор, пока код приложения не будет работать в его собственной службе, а данные, которыми он управляет, не будут извлечены в его собственную логически изолиро-

ванную базу данных. Но поскольку эта книга в основном посвящена поступательным изменениям, мы должны немного разведать вопрос, в какой последовательности это извлечение должно выполняться. У нас есть несколько вариантов:

- Сначала выделить базу данных, а затем код.
- Сначала выделить код, а затем базу данных.
- Выделить и то, и другое сразу.

У каждого варианта есть свои плюсы и минусы. Давайте сейчас рассмотрим эти варианты, а также некоторые шаблоны, которые помогут в зависимости от подхода, который вы принимаете.

Сначала выделить базу данных

С помощью отдельной схемы мы будем потенциально увеличивать число вызовов базы данных для выполнения одного-единственного действия. Если раньше, у нас, возможно, были все необходимые данные в одной инструкции `SELECT`, то теперь нам, возможно, потребуется вытаскивать данные из двух мест и соединять их в памяти. Кроме того, мы в итоге нарушаем транзакционную целостность, когда переходим к двум схемам, что окажет значительное влияние на наши приложения; мы обсудим эти проблемы позже в этой главе, когда рассмотрим такие темы, как распределенные транзакции и саги, и то, как они помогают. Выделяя схемы, но держа код приложения вместе, как показано на рис. 4.26, мы даем себе возможность откатывать наши изменения назад или продолжать корректировать вещи, не влияя на каких-либо потребителей нашей службы, если мы понимаем, что идем по неправильному пути. Как только мы убедимся, что выделение БД имеет смысл, то можем подумать о разделении кода приложения на две службы.

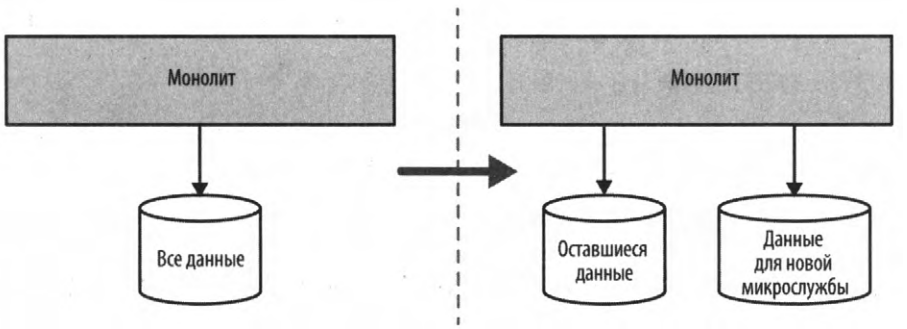


Рис. 4.26. Выделение сначала схемы позволяет раньше обнаружить проблемы с производительностью и транзакционной целостностью

Оборотная сторона состоит в том, что такой подход вряд ли принесет большую краткосрочную выгоду. Мы по-прежнему имеем монолитное развертывание кода. Можно утверждать, что головная боль от совместной базы данных ощущается только со временем, поэтому мы тратим время и усилия сейчас, чтобы получить

возврат в долгосрочной перспективе, не получая достаточную выгоду в краткосрочном плане. По этой причине я, скорее всего, пойду по этому пути, только если буду особо обеспокоен потенциальными проблемами производительности или согласованности данных. Также нужно учесть, что если сам монолит является черно-ящичной системой в рамках коммерческого софта, то описанный вариант нам недоступен.

Примечание по инструментарию

Вносить изменения в базы данных трудно по многим причинам. Одна из них — ограниченность набора инструментов, которые позволяли бы легко вносить изменения. В случае с кодом мы имеем инструменты рефакторизации, которые встроены в наши IDE, и у нас есть дополнительная выгода в том, что в своей основе системы, которые мы изменяем, не имеют состояния. В случае с базой данных изменяемые нами вещи имеют состояние, но нам также не хватает хорошего инструмента рефакторизации.

Много лет назад этот пробел в инструментах заставил меня и двух моих коллег, Ника Эшли (Nick Ashley) и Грэма Тэкли (Graham Tackley), создать инструмент с открытым исходным кодом под названием DBDeploy. Теперь несуществующий (создание инструмента с открытым исходным кодом очень отличается от его сопровождения!), он работал, позволяя улавливать изменения в SQL-скриптах, которые можно было выполнять детерминированным образом на схемах. В каждой схеме была специальная таблица, которая использовалась для отслеживания того, какие сценарии схемы были применены.

Цель DBDeploy состояла в том, чтобы позволить вам вносить поступательные изменения в схему, контролировать каждую версию изменений и разрешать исполнение изменений на нескольких схемах в разное время (подумайте о схемах этапов разработки, тестирования и производства).

В настоящее время я направляю людей к FlywayDB[11] или чему-то, что обеспечивает аналогичные возможности, но, какой бы инструмент вы ни выбрали, я настоятельно призываю вас убедиться, что он позволяет вам улавливать каждое изменение в дельта-скрипте, управляемом системой контроля версий.

Шаблон: "Один репозиторий на один ограниченный контекст"

В обычной практике принято иметь репозиторный слой, поддерживаемый каким-нибудь каркасом, таким, как Hibernate, для привязки вашего кода к базе данных, что упрощает отображение объектов или структур данных в базу данных и из нее. Вместо того чтобы иметь один репозиторный слой для всех наших задач доступа к данным, есть смысл подразделить эти репозитории по линиям ограниченных контекстов, как показано на рис. 4.27.

Наличие программного кода отображения базы данных, расположенного внутри кода для заданного контекста, помогает нам понять, какие части базы данных используются и какими фрагментами кода. Например, Hibernate делает это очень ясно, если у вас есть что-то вроде одного файла отображения на один ограниченный

[11] См. **https://flywaydb.org/**.

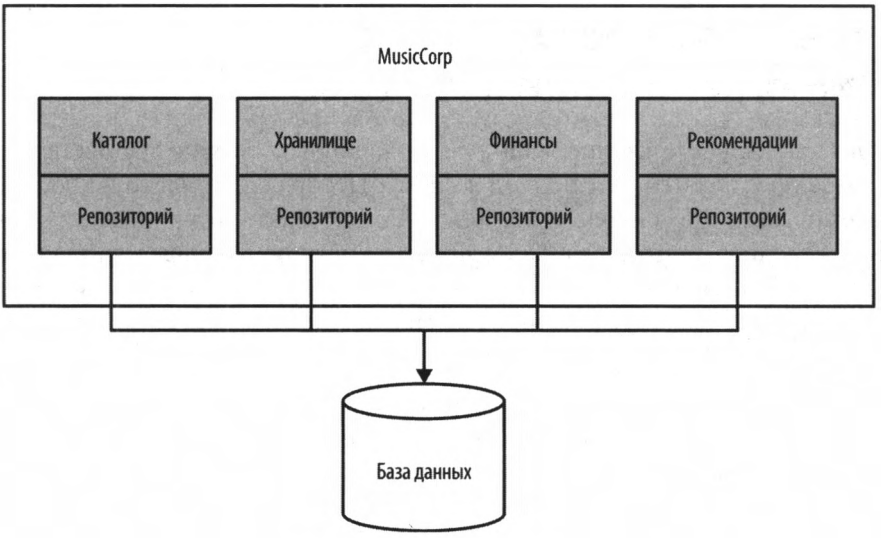


Рис. 4.27. Разбиение наших репозиторных слоев

контекст. Следовательно, мы видим, какие ограниченные контексты обращаются к каким таблицам в нашей схеме. Это помогает понять, какие таблицы необходимо перенести в рамках любой будущей декомпозиции.

Однако это не дает нам всей картины. Например, можно понять, что финансовый код использует таблицу "Приходно-расходный регистр" и каталожный код — таблицу "Товарные позиции", но далеко не всегда ясно, что база данных обеспечивает соблюдение связи по внешнему ключу из таблицы "Приходно-расходный регистр" в таблицу "Товарные позиции". Для того чтобы увидеть эти ограничения уровня базы данных, что бывает камнем преткновения, нам нужен еще один инструмент для визуализации данных. Отличное место для старта — использовать инструмент, подобный свободно доступному SchemaSpy[12], который генерирует графические представления связей между таблицами.

Все это помогает вам понять сопряженность между таблицами, которая охватывает то, что в итоге станет контурами служб. Но как оборвать эти связи? А как насчет случаев, когда одни и те же таблицы используются из нескольких ограниченных контекстов? Мы подробно рассмотрим эту тему далее в этой главе.

Где его использовать

Указанный шаблон действительно эффективен в любой ситуации, когда вы хотите переработать монолит, чтобы лучше понять, как его разделять. Деление этих репозиторных слоев по линиям доменных понятий поможет вам уяснить, где имеются стыки для микрослужб не только в вашей базе данных, но и в самом коде.

[12] См. **http://schemaspy.sourceforge.net/**.

Шаблон: "Одна база данных на один ограниченный контекст"

После того как вы четко изолировали доступ к данным с точки зрения приложения, имеет смысл продолжить этот подход в схеме. Центральное место в идее независимой развертываемости микрослужб занимает тот факт, что они должны владеть своими собственными данными. Прежде чем приступить к выделению кода приложения, мы можем начать эту декомпозицию, четко отделяя наши базы данных вокруг линий выявленных ограниченных контекстов.

В ThoughtWorks мы занимались имплементацией нескольких новых механизмом для расчета и прогнозирования валового дохода компании. В рамках этой работы мы выявили три широкие области функциональности, которые нужно было воплотить в коде. Я обсудил эту проблему с руководителем данного проекта Питером Гиллардом-Моссом (Peter Gillard-Moss). Питер объяснил, что функциональность ощущается совершенно отдельной, но у него есть опасения относительно лишней работы, которую принесет наличие этой функциональности, если она будет помещена в отдельные микрослужбы. В то время его группа была небольшой (всего три человека), и чувствовалось, что группа не могла оправдать выделение этих новых служб. В конце концов, они остановились на модели, в которой новая функциональность валового дохода была практически развернута как единая служба, содержащая три изолированных ограниченных контекста (каждый из которых оказался отдельным файлом JAR), как показано на рис. 4.28.

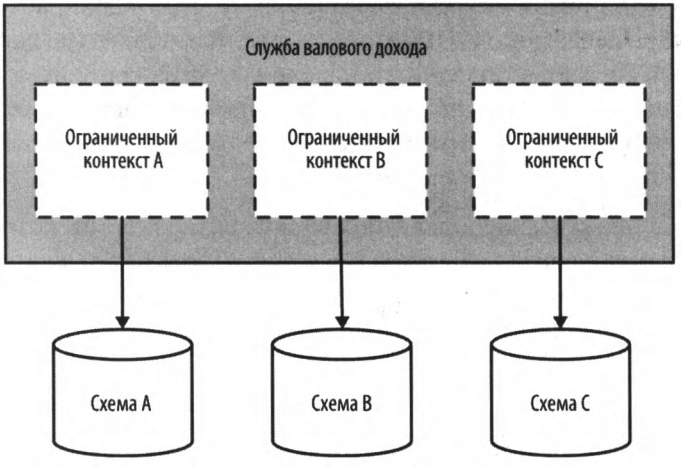


Рис. 4.28. Каждый ограниченный контекст в службе валового дохода имел свою собственную отдельную схему базы данных, допускающую разложение в дальнейшем

Каждый ограниченный контекст имел свои собственные, совершенно отдельные базы данных. Идея заключалась в том, что при необходимости в дальнейшем разложить базы данных на микрослужбы, это было бы намного проще. Однако оказалось, что такой необходимости никогда не было. Несколько лет спустя служба ва-

лового дохода остается такой, как есть, монолитом с многочисленными ассоциированными базами данных — отличным примером модульного монолита.

Где его использовать

На первый взгляд, лишняя работа по сопровождению отдельных баз данных не имеет большого смысла, если вы держите вещи в виде монолита. Я рассматриваю указанный шаблон как метод, который всецело строится на заботе о хеджировании ваших ставок. Он несет в себе немного больше работы, чем единая база данных, но держит ваши варианты открытыми относительно перехода на микрослужбы позже. Даже если вы никогда не перейдете на микрослужбы, четкое выделение схемы, поддерживающей базу данных, действительно поможет, в особенности, если у вас работает много людей над самим монолитом.

Именно этот шаблон я почти всегда рекомендую людям, строящим совершенно новые системы (в отличие от реимплементации существующей системы). Я не поклонник имплементации микрослужб для новых продуктов или стартапов. Ваше понимание домена вряд ли будет достаточно зрелым, для того чтобы знать, как выявлять стабильные границы домена. В случае стартапов, в особенности, поскольку природа вещи, которую вы строите, может резко измениться. Вместе с тем этот шаблон будет хорошей промежуточной точкой. Держите выделение схемы там, где, по вашему мнению, у вас в будущем будет выделение службы. Благодаря этому вы получаете некоторые выгоды от размежевания этих идей, одновременно снижая сложность системы.

Сначала выделить код

В общем случае я нахожу, что большинство групп сначала выделяют код, а затем базу данных, как показано на рис. 4.29. Они получают краткосрочное улучшение (надо надеяться) от новой службы, что дает им уверенность в завершении декомпозиции путем выделения базы данных.

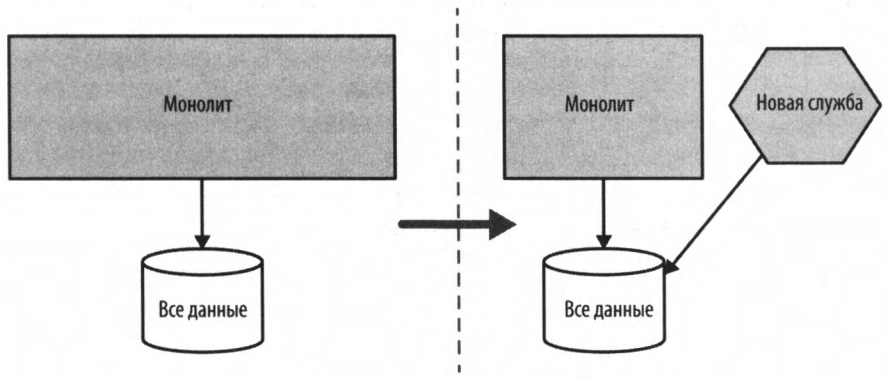


Рис. 4.29. Выделение сначала прикладного яруса оставляет нас с одной совместной схемой

В результате выделения прикладного яруса намного проще понять, какие данные необходимы новой службе. Вы также раньше получаете выгоду от наличия независимо развертываемого программного артефакта. Трудности, которые я всегда испытывал с этим подходом, заключаются в том, что группы нередко доходят до этого места и останавливаются, оставляя совместную базу данных в игре на постоянной основе. Если вы идете в этом направлении, то должны понимать, что если вы не доведете выделение до яруса данных, то вы накапливаете неприятности на будущее. Я видел группы, которые попадались в такую ловушку, но с радостью могу сообщить об организациях, которые здесь поступили правильно. JustSocial — одна из таких организаций, которая использовала описанный подход в рамках своей собственной миграции на микрослужбы. Другая потенциальная трудность здесь заключается в том, что вы, возможно, будете откладывать обнаружение неприятных сюрпризов, вызываемых "заталкиванием" операций соединения вверх в прикладной ярус.

Если вы идете в этом направлении, то будьте перед собой честны: уверены ли вы в том, что сможете обеспечить выделение любых данных, принадлежащих микрослужбе, в рамках следующего шага?

Шаблон: "Монолит как слой доступа к данным"

Вместо прямого доступа к данным из монолита мы можем просто перейти к модели, в которой мы создаем API в самом монолите. На рис. 4.30 службе "Счет-фактура" требуется информация о сотрудниках в нашей службе "Клиенты", поэтому мы создаем API "Сотрудники", позволяющий службе "Счет-фактура" к ним обращаться. Сюзанна Кайзер (Susanne Kaiser) из JustSocial поделилась со мной указанным шаблоном, т. к. эта компания успешно использовала его в рамках миграции

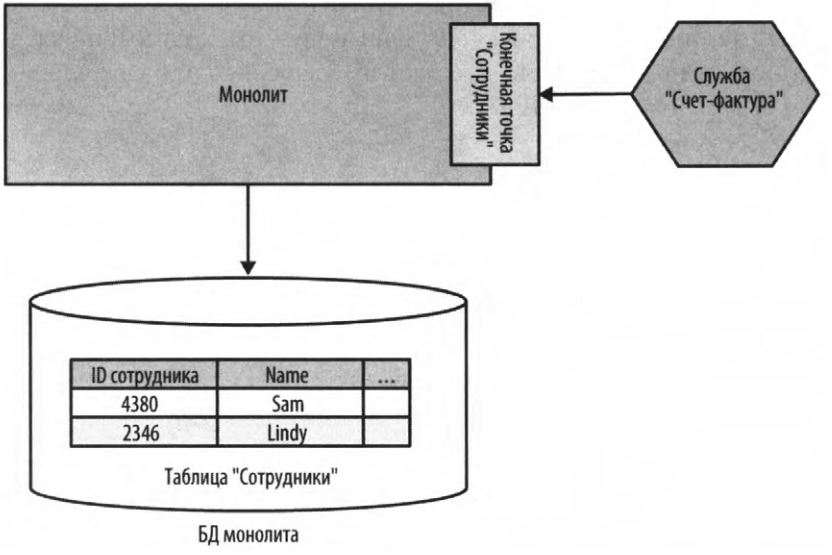
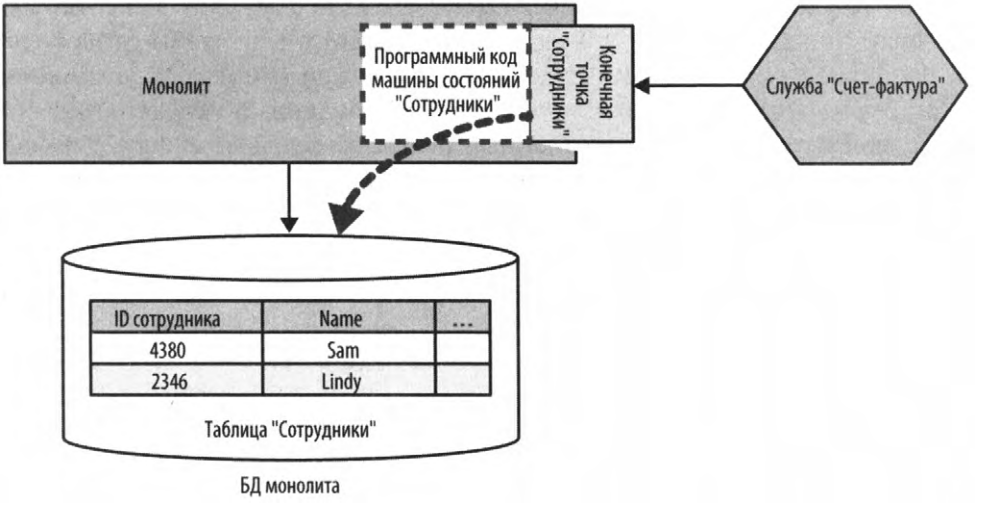


Рис. 4.30. Выставление API наружу на монолите позволяет службе избегать прямой привязки данных

До: Данные и функциональность о сотрудниках размещены в монолите, доступ к ним через API



После: Данные и функциональность о сотрудниках извлечены из монолита в новую микрослужбу

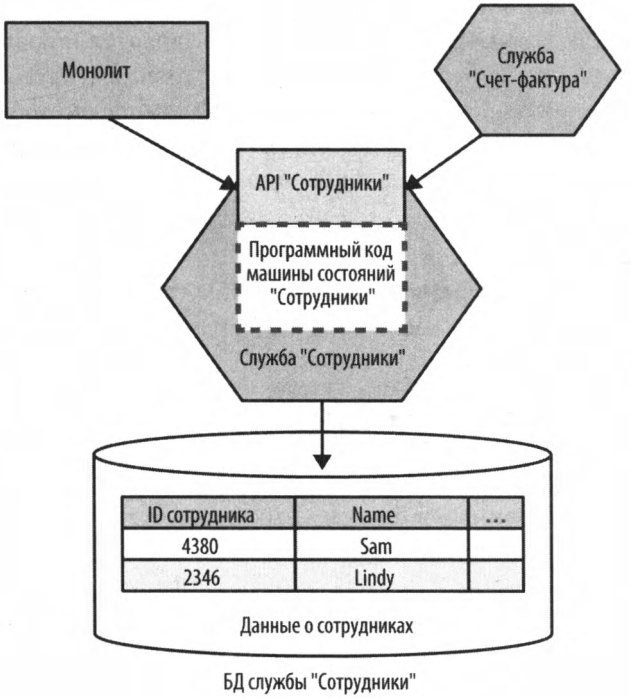


Рис. 4.31. Использование API "Сотрудники" для выявления контура службы "Сотрудники", которая должна быть выделена из монолита

на свои микрослужбы. Указанный шаблон имеет так много преимуществ, что я удивлен, что он не так известен, каким должен быть.

Одна из причин, почему указанный шаблон не применяется шире, вероятно, кроется в том, что у людей как бы застряла в голове мысль о том, что монолит мертв и бесполезен. Они хотят отойти от него. Они не думают о том, чтобы сделать его полезнее! Но плюсы здесь очевидны: нам (пока) не нужно заниматься декомпозицией данных, а необходимо заняться сокрытием информации, облегчая изолирование нашей новой службы от монолита. Я был бы более склонен принять эту модель, если бы чувствовал, что данные в монолите останутся там. Но этот шаблон хорошо работает, в особенности если вы думаете, что ваша новая служба практически не будет иметь состояния.

Не так уж сложно смотреть на этот шаблон как на способ выявления других кандидатных служб. Развивая эту идею, мы могли увидеть, что API "Сотрудники" выделяется из монолита, становясь микрослужбой самой по себе, как показано на рис. 4.31.

Где его использовать

Указанный шаблон лучше всего работает, когда код, управляющий данными, по-прежнему находится в монолите. Как мы уже говорили ранее, одним из способов думать о микрослужбах в том, что касается данных, является инкапсуляция состояния и кода, управляющего переходами из одного состояния в другое. Поэтому если переходы этих данных из одного состояния в другое по-прежнему обеспечиваются в монолите, то из этого следует, что микрослужба, которая хочет обратиться к этому состоянию (или изменить его), должна пройти через переходы из состояния в состояние в монолите.

Если данные, к которым вы пытаетесь обратиться в базе данных монолита, на самом деле должны быть "во владении" микрослужбы, то я больше склонен предложить пропустить этот шаблон и вместо него выделить данные.

Шаблон: "Мультисхемное хранение"

Как мы уже обсуждали, полезная идея — не усугублять плохую ситуацию. Если вы по-прежнему используете данные в базе данных напрямую, то это не означает, что новые данные, хранящиеся в микрослужбе, должны туда входить тоже. На рис. 4.32 приведен пример нашей службы "Счет-фактура". Стержневые счет-фактурные данные по-прежнему располагаются в монолите, откуда мы (в настоящее время) к ним обращаемся. Мы добавили возможность добавлять в счета-фактуры пересмотры. Это совершенно новая функциональность, находящаяся вне монолита. Для ее поддержки нам нужно хранить таблицу перепроверяющих лиц, отображая сотрудников в идентификаторы счетов-фактур. Если мы поместим эту новую таблицу в монолит, то будем помогать базе данных расти! Вместо этого мы поместили эти новые данные в нашу собственную схему.

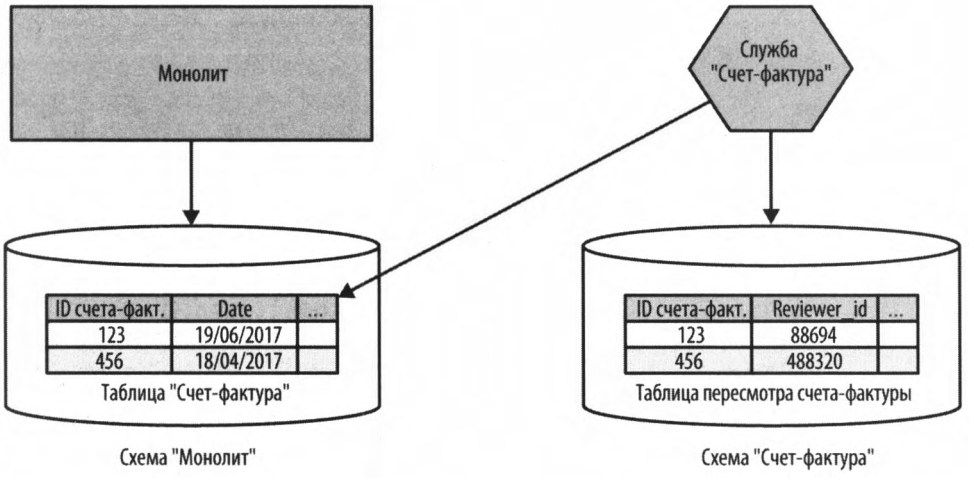


Рис. 4.32. Служба "Счет-фактура" помещает новые данные в свою собственную схему, но по-прежнему обращается к старым данным в монолите напрямую

В этом примере мы должны подумать о том, что произойдет, когда связь по внешнему ключу практически охватывает границу схемы. Позже в этой главе мы разведаем указанную проблему подробнее.

Вытаскивание данных из монолитной базы данных займет некоторое время, и, возможно, вы не сможете сделать это за один шаг. Поэтому вы должны быть вполне довольны тем, что ваша микрослужба имеет доступ к данным в монолитной БД, а также управляет своим собственным локальным хранением. Поскольку вы управляете перетаскиванием остальных данных из монолита, то можете выполнять миграцию в свою новую схему по одной таблице за раз.

Где его использовать

Указанный шаблон хорошо работает при добавлении совершенно новой функциональности в вашу микрослужбу, требующую хранения новых данных. Это явно не те данные, которые нужны монолиту (там нет функциональности), поэтому с самого начала держите их отдельно. Этот шаблон также имеет смысл, когда вы начинаете переносить данные из монолита в свою собственную схему, поскольку такой процесс может занимать некоторое время.

Если данные, к которым вы обращаетесь в схеме монолита, никогда не планируется переносить в свою схему, то я настоятельно рекомендую использовать шаблон "Монолит как слой доступа к данным" (см. *раздел "Шаблон: монолит как слой доступа к данным"*) в сочетании с описанным здесь шаблоном.

Выделить базу данных и код вместе

С точки зрения стадийности, конечно, у нас есть возможность просто выделить все за один большой шаг, как показано на рис. 4.33. Мы выделяем код и данные одним махом.

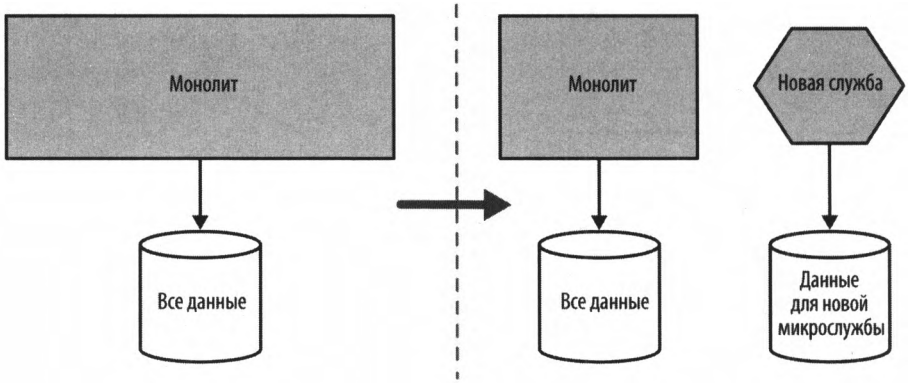


Рис. 4.33. Выделение кода и данных за один шаг

Здесь меня беспокоит то, что этот шаг гораздо обширнее, и пройдет больше времени, прежде чем вы сможете оценить влияние вашего решения в результате. Я настоятельно рекомендую избегать этого подхода и вместо него сначала выделять либо схему, либо прикладной ярус.

Так что же мне выделять сначала?

Понимаю: вы, наверное, устали от всех этих оговорок, дескать, "все зависит от конкретных условий", верно? И я не могу вас винить. Проблема в том, что у каждого своя ситуация, поэтому я пытаюсь вооружить вас достаточным контекстом и обсудить различные плюсы и минусы, чтобы помочь вам принять свое собственное решение. Тем не менее я знаю, что иногда люди хотят самую малость "горячего" взгляда на эти вещи, так что вот он.

Если я способен изменить монолит и если меня беспокоит потенциальное влияние на производительность или согласованность данных, то я сначала попытаюсь выделить схему. В противном случае я выделю код и буду использовать его для того, чтобы понять, как это влияет на владение данными. Но важно, чтобы вы также думали самостоятельно и учитывали любые факторы, которые повлияют на процесс принятия решений в вашей конкретной ситуации.

Примеры выделения схемы

До сих пор мы рассматривали выделение схемы на довольно высоком уровне, но есть ряд сложных проблем, связанных с декомпозицией базы данных, и несколько запутанных вопросов, которые требуется прояснить. Теперь мы рассмотрим несколько более низкоуровневых шаблонов декомпозиции данных и разведаем влияние, которое они оказывают.

Реляционные базы данных против NoSQL

Во многих примерах рефакторизаций, подробно описанных в этой главе, разведываются трудности, возникающие во время работы с реляционными схемами. Природа этих типов баз данных создает дополнительные трудности с точки зрения выделения

схем. Многие из вас вполне могут пользоваться альтернативными типами нереляционных баз данных. Тем не менее многие из приведенных далее шаблонов по-прежнему могут применяться. У вас, возможно, будет меньше ограничений в том, как вносить изменения, но я надеюсь, что данный совет будет по-прежнему полезным.

Шаблон: "Разложить таблицу"

Иногда вы будете находить данные в одной таблице, которые нуждаются в разделении вдоль двух или более контуров служб, и этот вопрос представляет интерес. На рис. 4.34 мы видим одну совместную таблицу "Товарная позиция", в которой хранится информация не только о продаваемой товарной позиции, но и об уровнях ее запасов.

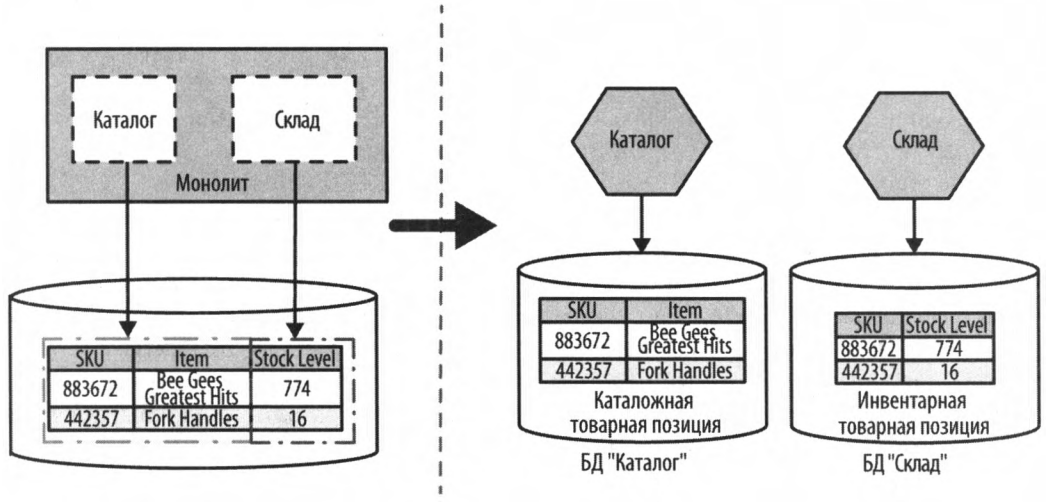


Рис. 4.34. Одна таблица, соединяющая в себе два выделяемых ограниченных контекста

В рассматриваемом примере мы хотим выделить каталог и склад как новые службы, но данные для них перемешаны в одной таблице. Поэтому нам нужно разложить эти данные на две отдельные таблицы, как показано на рис. 4.34. В духе поступательной миграции, возможно, следует отделить таблицы друг от друга в существующей схеме перед выделением схем. Если эти таблицы существовали в одной схеме, то имеет смысл объявить связь по внешнему ключу из столбца "Артикул инвентарной товарной позиции" (SKU) в таблицу "Каталожная товарная позиция". Однако, поскольку в итоге мы планируем перенести эти таблицы в отдельные базы данных, мы, возможно, не извлечем из этого большой выгоды, поскольку у нас не будет единой базы данных, обеспечивающей согласованность данных (вскоре мы рассмотрим эту идею подробнее).

Этот пример довольно прямолинейный, отделить владение данными, один столбец за другим, было легко. Но что произойдет, когда несколько частей кода обновляют один и тот же столбец? На рис. 4.35 мы имеем таблицу "Клиент", которая содержит столбец "Статус".

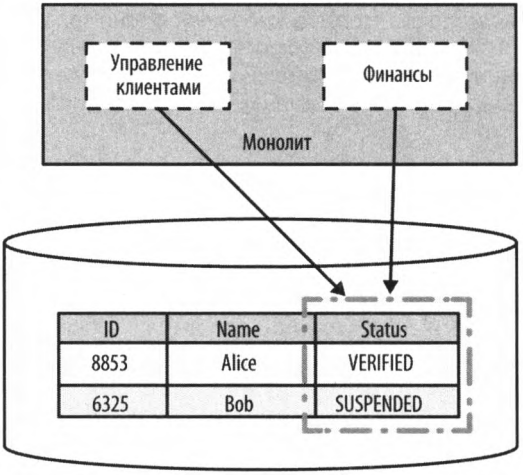


Рис. 4.35. И код "Управления клиентами", и код "Финансов" могут изменять статус в таблице "Клиент"

Этот столбец обновляется во время регистрации клиента, указывая на то, что данное лицо подтвердило (или не подтвердило) свою электронную почту, при этом значение переходит из НЕ_ВЕРИФИЦИРОВАНО в ВЕРИФИЦИРОВАНО. После того как клиент ВЕРИФИЦИРОВАН, он может делать покупки. Наш финансовый код занимается приостановкой клиентов, если их счета не оплачены, поэтому они иногда меняют статус клиента на ПРИОСТАНОВЛЕН. В этом случае "Статус" клиента по-прежнему выглядит так, как будто он должен быть частью доменной модели клиента, и как таковой он должен управляться службой "Клиенты", которая вскоре будет создана. Помните, что мы хотим, где это возможно, держать машины состояний для сущностей нашего домена внутри контура одной службы, и обновление "Статуса", безусловно, ощущается для клиента как часть машины состояний! Это означает, что после выделения служб наша новая служба "Финансы" будет нуждаться в вызове службы для обновления этого статуса, как мы видим на рис. 4.36.

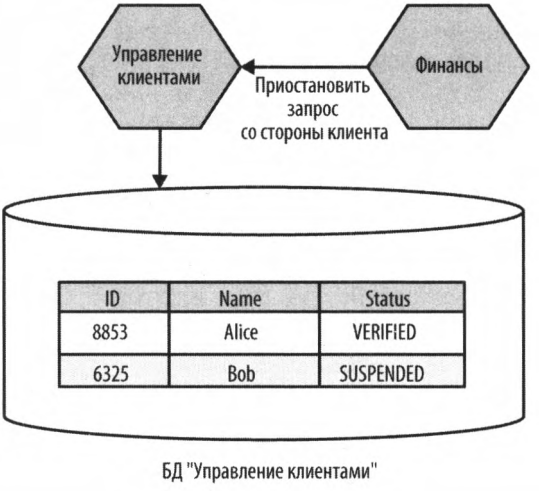


Рис. 4.36. Новая служба "Финансы" должна совершать вызовы службы для приостановки клиента

Большая проблема с таким разложением таблиц заключается в том, что мы теряем безопасность, предоставляемую нам транзакциями базы данных. Мы разведаем эту тему подробнее в конце этой главы, когда в соответствующих разделах рассмотрим "Транзакции" и "Саги".

Где его использовать

На первый взгляд указанный шаблон кажется довольно простым. Если таблица принадлежит двум или более ограниченным контекстам в текущем монолите, то вам нужно разложить таблицу по этим линиям. Если вы находите конкретные столбцы в таблице, которые, по-видимому, обновляются многочисленными частями вашей кодовой базы, то вам нужно принять решение на свое усмотрение в отношении того, кто должен "владеть" этими данными. Находится ли в центре вашего внимания существующее доменное понятие? Ответ на этот вопрос поможет определить, куда эти данные должны пойти.

Шаблон: "Перенести связь по внешнему ключу в код"

Мы решили извлечь нашу службу "Каталог", которая может управлять и предоставлять информацию об исполнителях, треках и альбомах. В настоящее время наш каталожный программный код внутри монолита хранит информацию о компакт-дисках, которые у нас имеются для продажи в таблице "Альбомы". Эти альбомы в итоге будут упомянуты в нашей таблице "Приходно-расходный регистр", где мы отслеживаем все продажи. Вы видите это на рис. 4.37. Строки в таблице "Приходно-расходный регистр" просто регистрируют сумму, которую мы получаем за компакт-диск, вместе с идентификатором, который ссылается на проданную товарную позицию. Идентификатор в нашем примере называется SKU[13] — распространенная практика в розничных системах.

В конце каждого месяца мы должны составлять отчет с описанием наших самых продаваемых компакт-дисков. Таблица "Приходно-расходный регистр" помогает нам понять, копий какого артикула продано больше всего, но информация об этом артикуле находится в таблице "Альбомы". Мы хотим, чтобы отчеты были приятными и удобными для чтения, поэтому вместо утверждения: "Мы продали 400 копий артикула 123 и заработали 1596 долларов", мы хотели бы видеть более информативную фразу: "Мы продали 400 копий Born To Run Брюса Спрингстина и заработали 1596 долларов". Для этого запрос к базе данных, инициированный нашим финансовым кодом, должен присоединить информацию из таблицы "Приходно-расходный регистр" к таблице "Альбомы", как показано на рис. 4.37.

Мы определили связь по внешнему ключу в нашей схеме так, что строка в таблице "Приходно-расходный регистр" идентифицируется как имеющая связь со строкой

[13] SKU (stock keeping unit) — это артикул, или единица складского учета, инвентарной позиции — *Пер.*

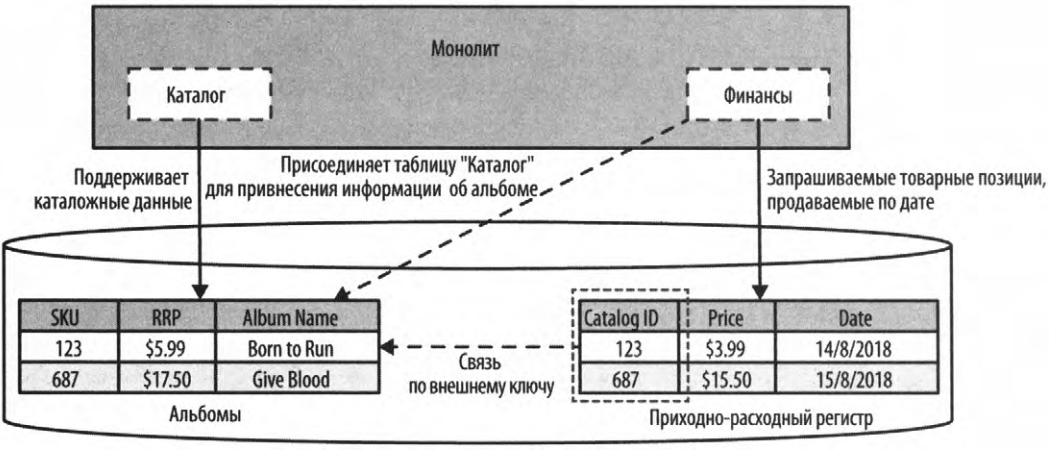


Рис. 4.37. Связь по внешнему ключу

в таблице "Альбомы". Определяя такую связь, опорное ядро СУБД способно обеспечивать согласованность данных: если строка в таблице "Приходно-расходный регистр" ссылается на строку в таблице "Альбомы", то мы знаем, что эта строка существует. В нашей ситуации это означает, что мы всегда можем получить информацию о проданном альбоме. Такие связи по внешнему ключу также позволяют ядру СУБД выполнять оптимизацию производительности для обеспечения максимально быстрого выполнения операции соединения.

Мы хотим выделить каталожный и финансовый код в собственные соответствующие им службы, и это означает, что данные должны идти тоже. Если таблицы "Альбомы" и "Приходно-расходный регистр" в итоге будут располагаться в разных схемах, то что произойдет с нашей связью по внешнему ключу? Скажем так, нам придется подумать о двух ключевых проблемах. Во-первых, когда наша новая служба "Финансы" захочет сгенерировать этот отчет в будущем, то как она будет извлекать каталожную информацию, если не сможет этого сделать через существующую в базе данных операцию соединения? Во-вторых, что мы сделаем с тем фактом, что в нашем новом мире теперь будет существовать несогласованность данных?

Перенос операции соединения

Давайте сначала рассмотрим замену соединения (операции join). В случае монолитной системы для присоединения строки таблицы "Альбомы" к информации о продажах в "Приходно-расходном регистре" база данных выполнила бы соединение за нас. Нам потребовался бы один-единственный запрос `SELECT`, и мы бы присоединились к таблице "Альбомы". Для исполнения запроса и извлечения всех необходимых данных необходим всего один вызов базы данных.

В нашем мире, основанном на микрослужбах, новая служба "Финансы" несет ответственность за генерирование отчета о бестселлерах, но не имеет данных об альбомах локально. Поэтому она должна будет извлечь их из новой службы "Каталог",

как мы видим на рис. 4.38. Во время генерирования отчета служба "Финансы" сначала запрашивает таблицу "Приходно-расходный регистр", извлекая список наиболее продаваемых артикулов за последний месяц. В этой точке у нас есть только список артикулов и число проданных копий по каждому; это единственная информация, которую мы имеем локально.

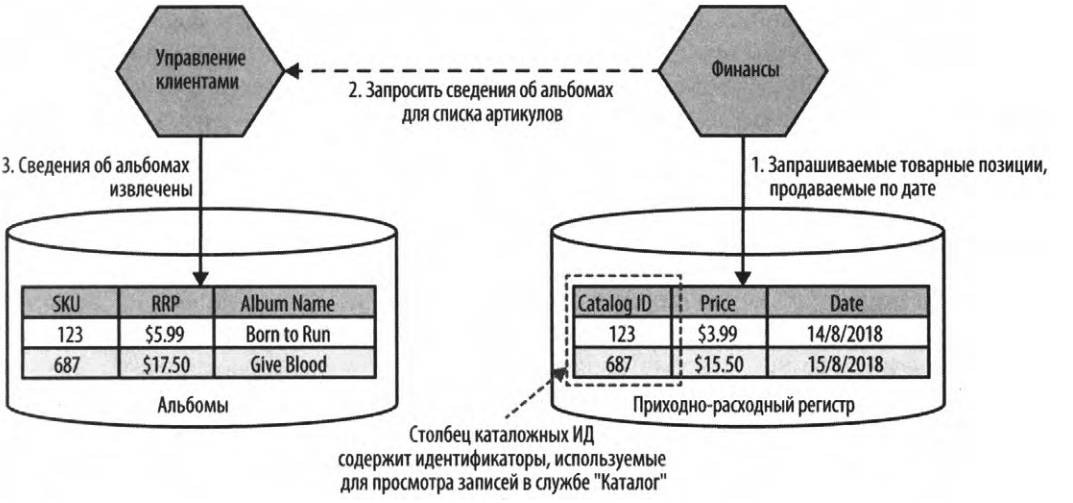


Рис. 4.38. Замена операции соединения базы данных вызовами служб

Далее нам нужно вызвать службу "Каталог", запросив информацию по каждому из этих артикулов. Этот запрос, в свою очередь, побудит службу "Каталог" сделать свой локальный запрос `SELECT` в своей собственной базе данных.

Логически операция соединения по-прежнему происходит, но теперь она выполняется внутри службы "Финансы", а не в базе данных. К сожалению, она не будет такой же эффективной, как раньше. Мы перешли из мира, где у нас есть одна инструкция `SELECT`, в новый мир, где у нас есть запрос `SELECT` к таблице "Приходно-расходный регистр", за которым следует вызов службы "Каталог", которая, в свою очередь, запускает инструкцию `SELECT` в отношении таблицы "Альбомы", как мы видим на рис. 4.38.

В этой ситуации я был бы очень удивлен, если совокупная задержка этой операции не увеличилась. Она не является существенной проблемой в данном конкретном случае, поскольку отчет генерируется ежемесячно и поэтому может агрессивно кэшироваться. Но если эта операция выполняется часто, то задержка станет проблемой. Мы можем смягчить вероятное влияние этого увеличения задержки, разрешив массовый поиск артикулов в службе "Каталог" или, возможно, даже локально кэшируя требуемую информацию об альбомах.

В конечном счете, является ли увеличение задержки проблемой — решать только вам. Вам нужно иметь представление о допустимой задержке для ключевых операций и уметь измерять текущую задержку. Распределенные системы, такие, как Jaeger, действительно здесь помогают, поскольку они предоставляют возможность

получать точный хронометраж операций, охватывающих несколько служб. Замедление операции может быть приемлемым, если она по-прежнему достаточно быстрая, в особенности если в качестве компромисса вы получаете некоторые другие выгоды.

Согласованность данных

Более запутанное соображение — возможность столкнуться с несогласованностью данных в случае отдельных служб "Каталог" и "Финансы" с отдельными схемами. С одной схемой я бы не смог удалить строку в таблице "Альбомы", если бы в таблице "Приходно-расходный регистр" была ссылка на эту строку. Моя схема обеспечила бы согласованность данных. В нашем новом мире такого соблюдения не существует. Какие у нас тут варианты?

Проверять перед удалением

Наш первый вариант заключается в том, чтобы при удалении записи из таблицы "Альбомы" сверяться со службой "Финансы", чтобы убедиться, что она еще не имеет ссылки на запись. Проблема здесь в том, что трудно гарантировать, что мы сможем сделать это правильно. Скажем, мы хотим удалить артикул 683. Мы посылаем вызов в "Финансы", говоря "Вы действительно используете 683?" Оттуда приходит ответ, что эта запись не используется. Затем мы удаляем запись, но пока мы это делаем, в системе "Финансы" создается новая ссылка на 683. Для того чтобы этого не произошло, нам нужно остановить создание новых ссылок на запись 683 до тех пор, пока не произошло удаление — что, вероятно, потребует установления замков и всех трудностей, которые следуют из распределенной системы.

Еще одна проблема с проверкой того, не используется ли уже та или иная запись, состоит в том, что она создает де-факто обратную зависимость от службы "Каталог". Теперь нам нужно будет сверяться с любой другой службой, которая использует наши записи. Уже плохо, если у нас информация необходима лишь одной-другой службе, но ситуация значительно ухудшается по мере того, как у нас становится больше потребителей.

Я настоятельно призываю вас не рассматривать этот вариант из-за трудности в обеспечении правильной имплементации этой операции, а также высокой степени сопряженности службы, которую она создает.

Улаживать удаление изящно

Более приемлемый вариант — предоставить службе "Финансы" возможность разбираться с тем фактом, что служба "Каталог" может не иметь информации об альбоме, делая это изящным способом. Это так же просто, как показать в нашем отчете сообщение "Информация об альбоме отсутствует", если мы не можем найти данный артикул.

В этой ситуации служба "Каталог" могла бы выдать нам сообщение, когда мы запрашиваем артикул, который раньше существовал. Например, здесь хорошо подошел бы код отклика под номером 410 GONE (Перестал существовать) при исполь-

зовании HTTP. Код отклика 410 отличается от обычного 404. 404 означает, что запрошенный ресурс не найден, тогда как 410 означает, что запрошенный ресурс был по указанному адресу, но больше недоступен. Это различие может быть важным, в особенности во время отслеживания проблем несогласованности данных! Даже если вы не применяете протокол на базе HTTP, подумайте о том, получите ли вы выгоду от поддержки такого отклика.

Если бы мы действительно хотели продвинуться, то могли бы обеспечить, чтобы наша служба "Финансы" информировалась, когда товарная позиция "Каталога" удаляется, возможно, путем подписки на события. Когда мы подхватываем событие удаления из "Каталога", мы могли бы решить скопировать информацию о теперь удаленном "Альбоме" в нашу локальную базу данных. В данной конкретной ситуации указанная мера кажется излишней, но будет полезной в других сценариях, в особенности если вы хотели бы имплементировать распределенную машину состояний для выполнения чего-то вроде каскадного удаления через контуры служб.

Не разрешать удаление

Один из способов уменьшения несогласованности в системе мог бы заключаться в том, чтобы просто не позволять удалять записи в службе "Каталог". Если в существующей системе удаление товарной позиции было сродни обеспечению недоступности товара для продажи или чему-то подобному, то мы могли бы просто имплементировать возможность "мягкого" удаления. Это можно сделать, например, с помощью столбца статуса, помечая данную строку как недоступную или, возможно, даже перенося строку в выделенную "кладбищенскую" таблицу[14]. В такой ситуации запись об альбоме по-прежнему может запрашиваться службой "Финансы".

И как же тогда обрабатывать удаление?

В сущности, мы создали режим сбоя, который не мог существовать в нашей монолитной системе. Разглядывая пути решения этого вопроса, мы должны учитывать потребности наших пользователей, поскольку разные решения могут влиять на наших пользователей по-разному. Выбор правильного решения здесь требует понимания вашего конкретного контекста.

Лично я бы в этой конкретной ситуации, скорее всего, решил это двумя путями: не позволяя удалять информацию об "Альбоме" в "Каталоге" и обеспечивая, чтобы службы "Финансы" занималась недостающей записью. Вы можете утверждать, что если запись нельзя удалить из службы "Каталог", то поиск из "Финансов" никогда не был бы безуспешным. Однако существует вероятность того, что в результате повреждения служба "Каталог" будет восстановлена в более раннее состояние, означающее, что запись, которую мы ищем, больше не существует. В этой ситуации я бы не хотел, чтобы служба "Финансы" рухнула. Этот набор обстоятельств

[14] Поддержка "исторических" данных в реляционной базе данных, такой, как эта, усложняется, в особенности если вам нужно программно воссоздавать старые версии ваших сущностей. Если у вас строгие требования в этой области, то было бы полезным разведывание источников событий как альтернативного способа поддержания состояния.

выглядит маловероятным, но я всегда ищу возможности усиления отказоустойчивости, и подумайте о том, что произойдет, если вызов не сработает; изящное улаживание этой ситуации в службе "Финансы" выглядит довольно легким исправлением.

Где его использовать

Когда вы начинаете думать об эффективном разрыве связей по внешнему ключу, одна из первых мер, которую вам нужно обеспечить, — это не разъединить две вещи, которые на самом деле хотят быть единым целым. Если вы беспокоитесь о том, что разъединяете агрегат, то сделайте паузу и пересмотрите свое решение. В случае "Приходно-расходного регистра" и "Альбомов" здесь кажется ясно, что у нас два отдельных агрегата со связью между ними. Но возьмем другой случай: таблицу "Заказ" и много ассоциированных строк в таблице "Строки заказа", содержащих сведения о заказанных товарных позициях. Если мы выделим "Строки заказа" в отдельную службу, то нам придется подумать о вопросах целостности данных. На самом деле строки заказа являются частью самого заказа. Следовательно, нам следует видеть их как единое целое, и если мы хотим перенести их из монолита, то их необходимо переносить вместе.

Иногда, беря более крупный "кусок" из монолитной схемы, вы будете в состоянии перенести с собой обе стороны связи по внешнему ключу, что намного упрощает вашу жизнь!

Пример: совместные статические данные

Статические справочные данные (которые меняются нечасто, но, как правило, имеют критически важный характер) приводят к некоторым интересным трудностям, и я видел несколько подходов к управлению ими. Чаще всего, они проявляют себя в базах данных. Я встречал, возможно, столько же кодов стран, которые хранятся в базах данных (показано на рис. 4.39), сколько я написал своих собственных классов `String Utils` для проектов на Java.

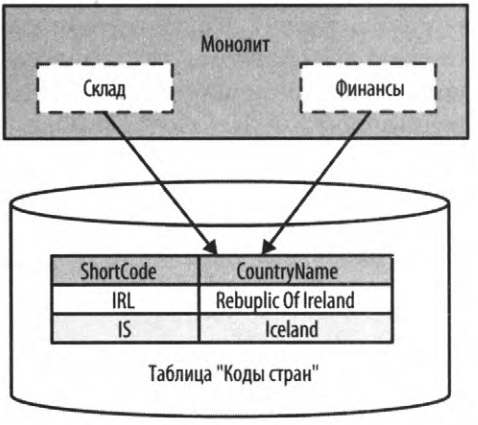


Рис. 4.39. Коды стран в базе данных

Я всегда задавался вопросом, почему малые объемы редко изменяющихся данных, таких, как коды стран, должны существовать в базах данных, но независимо от первопричины эти примеры совместных статических данных, хранимых в базах данных, возникают часто. Тогда что делать, если в нашем музыкальном магазине многие части программного кода нуждаются в одних и тех же статических справочных данных? Выясняется, у нас тут большой выбор вариантов.

Шаблон: "Дублировать статические справочные данные"

Почему бы просто не предоставить каждой службе свою собственную копию данных, как показано на рис. 4.40? Вероятно, у многих из вас от неожиданности это вызовет резкий вдох. Дублировать данные? Я сошел с ума? Послушайте! Все не так безумно, как вы думаете.

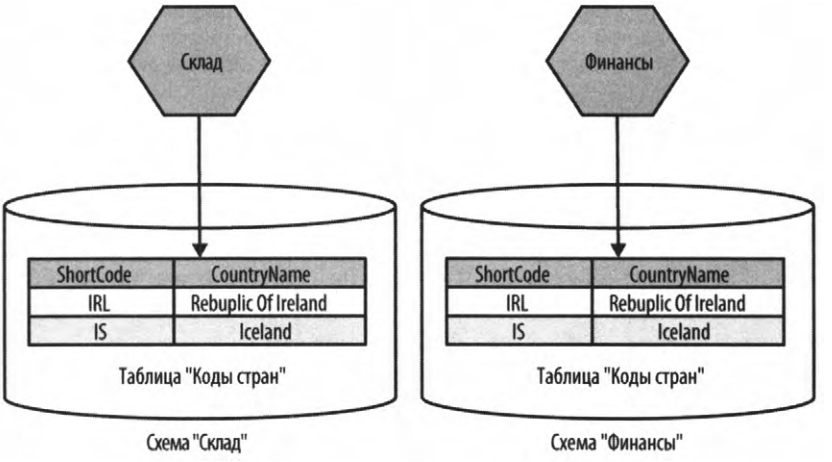


Рис. 4.40. Каждая служба имеет свою собственную схему "Коды стран"

Обеспокоенности, связанные с дублированием данных, как правило, сводятся к двум вещам. Во-первых, каждый раз, когда мне нужно изменить данные, я должен делать это в нескольких местах. Но как часто в описываемой ситуации данные меняются? Последний раз, когда страна была создана и официально признана, было в 2011 году, с появлением Южного Судана (короткий код которого, SSD). Поэтому я не думаю, что тут есть, о чем волноваться, верно? Больше беспокоит то, что произойдет, если данные будут несогласованными? Например, служба "Финансы" знает, что Южный Судан — это страна, но необъяснимым образом служба "Склад" живет в прошлом и ничего о нем не знает.

Вопрос о том, вызывает или нет несогласованность затруднения, сводится к варианту использования данных. В рассматриваемом примере учтите, что "Склад" использует данные кодов стран для регистрации места производства наших компакт-дисков. Оказывается, что у нас на складе нет компакт-дисков производства Южного Судана, поэтому тот факт, что мы не располагаем этими данными, не влечет ни-

каких затруднений. С другой стороны, служба "Финансы" нуждается в информации о кодах стран для регистрации информации о продажах, и у нас есть клиенты в Южном Судане, поэтому нам нужна эта обновленная запись.

Когда данные необходимы только локально внутри каждой службы, несогласованность не является проблемой. Вспомните наше определение ограниченного контекста: все дело в том, что информация скрыта внутри контуров. Если, с другой стороны, данные составляют часть коммуникации между этими службами, то у нас будут другие поводы для беспокойства. Если и "Складу", и "Финансам" нужен одинаковый вид информации о стране, то дублирование такого рода определенно меня беспокоит.

Конечно, мы также могли бы подумать о синхронизации этих копий с помощью какого-то фонового процесса. В такой ситуации мы вряд ли гарантируем, что все копии будут согласованы, но если исходить из того, что наш фоновый процесс выполняется достаточно часто (и быстро), то мы можем уменьшить потенциальное окно несогласованности в наших данных, и это будет достаточно хорошо.

Как разработчики, мы часто плохо реагируем, когда видим дублирование. Мы беспокоимся о лишней стоимости управления повторными копиями информации и еще больше волнуемся, если эти данные расходятся. Но иногда дублирование — это меньшее из двух зол. Принятие некоторого дублирования в данных будет разумным компромиссом, если оно означает, что мы избегаем введения сопряженности.

Где его использовать

Указанный шаблон следует использовать лишь изредка, и вы должны предпочесть ему некоторые варианты, которые мы рассмотрим позже. Он иногда полезен для крупных объемов данных, когда не обязательно, чтобы все службы видели один и тот же набор данных. Хорошим вариантом будет что-то вроде файлов почтовых индексов в Великобритании, где вы периодически получаете обновления отображений из почтовых индексов в адреса. Это довольно крупный набор данных, управлять которым в кодовой форме, вероятно, будет болезненно. Если вы хотите присоединиться к этим данным напрямую, то целесообразно выбрать указанный вариант, но я буду честен и скажу, что не помню, чтобы я когда-либо делал это сам!

Шаблон: "Выделенная схема справочных данных"

Если вам действительно нужен один источник истины для кодов стран, то вы можете перенести эти данные в выделенную схему, возможно, отложенную в сторону для всех статических справочных данных, как мы видим на рис. 4.41.

Нам все-таки придется подумать обо всех трудностях совместной базы данных. В какой-то степени обеспокоенность относительно сопряженности и изменения несколько компенсируется природой данных. Они изменяются нечасто и структурированы просто, и, следовательно, мы могли бы легче смотреть на эту схему справочных данных как на определенный интерфейс. В описанной ситуации я бы смот-

рел на эту схему справочных данных как на ее собственную версионную сущность и обеспечил, чтобы люди понимали, что структура схемы представляет для потребителей интерфейс службы. Внесение переломных изменений в эту схему, судя по всему, будет болезненным.

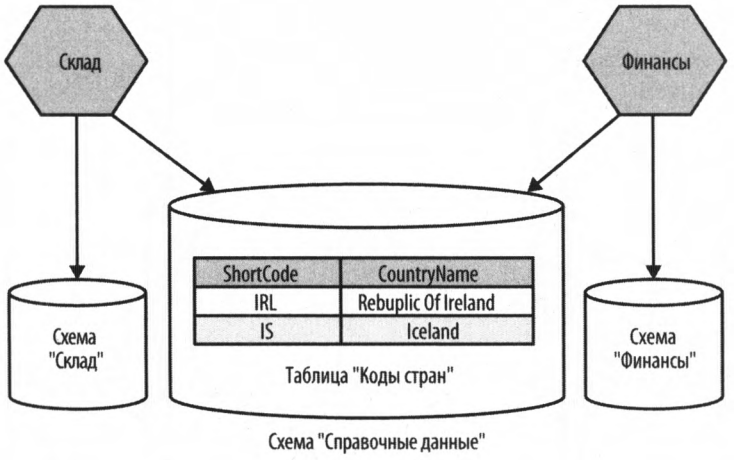


Рис. 4.41. Использование выделенной совместной схемы для справочных данных

Наличие данных в схеме открывает возможность их использования службами в рамках запросов на соединение к своим собственным локальным данным. Однако для этого вам, вероятно, потребуется обеспечить, чтобы схемы были расположены на одном и том же опорном ядре СУБД. Это добавляет некоторую сложность с точки зрения отображения логического "мира" в физический, игнорируя потенциальные проблемы с единственной точкой сбоя.

Где его использовать

Указанный вариант имеет много достоинств. Мы избегаем опасений, связанных с дублированием, и формат данных вряд ли изменится, поэтому некоторые из наших возможных опасений смягчаются. Этот подход допустим для крупных объемов данных или для варианта перекрестно-схемных соединений. Просто помните, что любые изменения формата схемы, скорее всего, значительно повлияют на многочисленные службы.

Шаблон: "Библиотека статических справочных данных"

Если присмотреться, то в наших данных кодов стран не так много записей. Из стандартного ISO-листинга мы видим, что в нем представлено только 249 стран[15].

[15] Для вас, поклонников ISO, речь идет об ISO 3166-1!.

Это будет хорошо вписываться в код, возможно, как простой статический перечисляемый тип. На самом деле, хранение малых объемов статических справочных данных в коде я делал много раз, и это, по моему опыту, пригодно для архитектур на основе микрослужб.

Конечно, мы бы предпочли не дублировать данные, если нам это не нужно, а это приводит нас к тому, чтобы подумать о размещении данных в библиотеке, которая будет статически связана с любыми службами, желающими получить эти данные. Stitch Fix, американский ритейлер модной одежды, часто использует такие совместные библиотеки для хранения статических справочных данных.

Рэнди Шоуп (Randy Shoup), бывший вице-президент по инженерным вопросам в Stitch Fix, высказался о том, что наиболее благоприятной зоной для этого метода были малые по объему типы данных, которые изменялись нечасто или вообще не менялись (и если они изменялись, то у вас было много предварительных предупреждений об изменении). Наглядный пример — классические обозначения размеров одежды: XS, S, M, L, XL для универсальных размеров или измерений внутреннего шва для брюк.

В нашем случае мы определяем отображения кодов стран в перечислимом типе `Country` и укомплектовываем их в библиотеку для использования в наших службах, как показано на рис. 4.42.

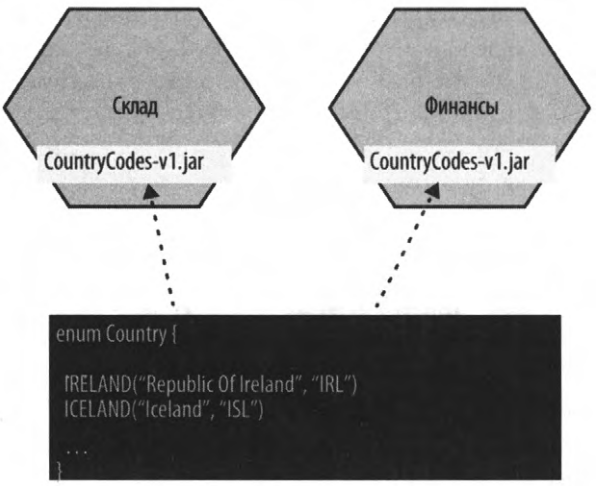


Рис. 4.42. Хранение справочных данных в библиотеке, которую можно использовать между службами совместно

Это решение является изящным, но оно не лишено недостатков. Очевидно, что если у нас есть смесь стеков технологий, то мы не сможем использовать одну совместную библиотеку. Кроме того, помните "золотое" правило микрослужб? Мы должны обеспечивать, чтобы наши микрослужбы развертывались независимо. Если бы нам потребовалось обновить библиотеку кодов стран и все службы должны были немедленно получить новые данные, то нам пришлось бы переразвернуть все службы в момент, когда новая библиотека будет доступна. Это классический командно-

строевой релиз[16], и именно такой ситуации мы пытаемся избежать с помощью архитектур на основе микрослужб.

На практике, если нам нужно, чтобы одни и те же данные были доступны везде, то поможет надлежащее уведомление об изменении. Примером, который привел Рэнди, была потребность добавить новый цвет в один из наборов данных Stitch Fix. Это изменение необходимо было развернуть для всех служб, которые использовали этот тип данных, но они имели значительное упредительное время на то, чтобы все группы забрали последнюю версию. Если вы возьмете пример кодов стран, опять-таки, у нас, вероятно, будет много предварительных сведений о необходимости добавить новую страну. Например, Южный Судан стал независимым государством в июле 2011 года после референдума, проведенного шестью месяцами ранее, давая нам много времени на то, чтобы осуществить наши изменения. Новые страны редко создаются по прихоти!

Если ваши микрослужбы используют совместные библиотеки, то помните, что вы должны смириться с тем, что у вас в производстве могут быть развернуты разные версии библиотеки!

Это означает, что при необходимости обновить библиотеку кодов стран нам придется согласиться с тем, что не все микрослужбы могут гарантированно иметь одну и ту же версию библиотеки, как показано на рис. 4.43. Если это для вас не работает, то, возможно, поможет следующий вариант.

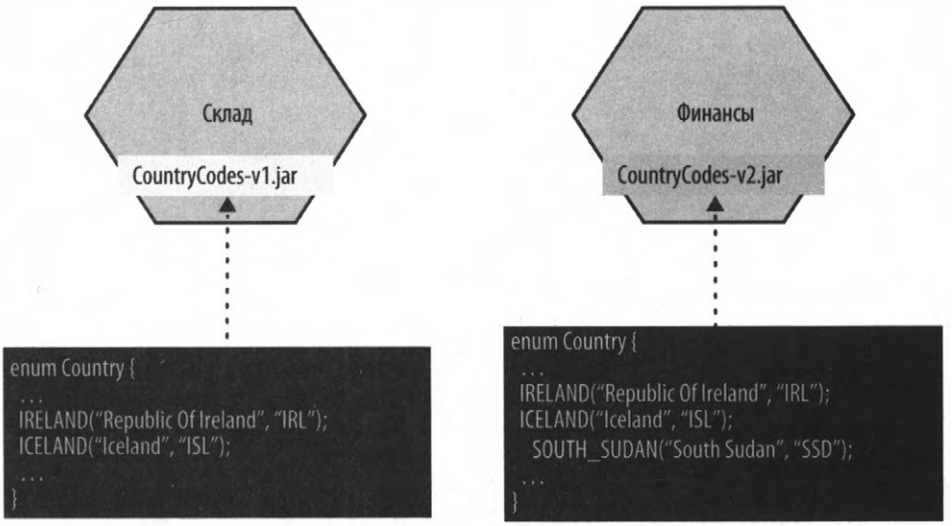


Рис. 4.43. Различия между совместными библиотеками справочных данных могут вызвать затруднения

[16] Командно-строевой релиз (lock-step release) в сфере программно-информационного обеспечения означает строительство отказоустойчивых систем путем обеспечения избыточности, когда выполняется один и тот же набор операций в одно и то же время параллельно. Образно обозначает стандартный метод или процедуру, которой придерживаются бездумно или которая сводит к минимуму индивидуальность. Термин lock-step взят из армейского лексикона, где он означает движение строевым шагом, слаженно в ногу — *Пер.*

В простом варианте этого шаблона затрагиваемые данные хранятся в конфигурационном файле, возможно, в стандартном файле свойств или, если требуется, в более структурированном формате JSON.

Где его использовать

Для небольших объемов данных, где вы можете расслабиться по поводу того, что разные службы видят разные версии этих данных, описанный вариант является отличным, но часто упускаемым из виду. Наличие видимости относительно того, какая служба какую версию данных имеет, особенно полезно.

Шаблон: "Служба статических справочных данных"

У меня есть подозрения, что вы понимаете, куда это все клонится. Эта книга посвящена созданию микрослужб, так почему бы не подумать о создании выделенной службы только для кодов стран, как показано на рис. 4.44?

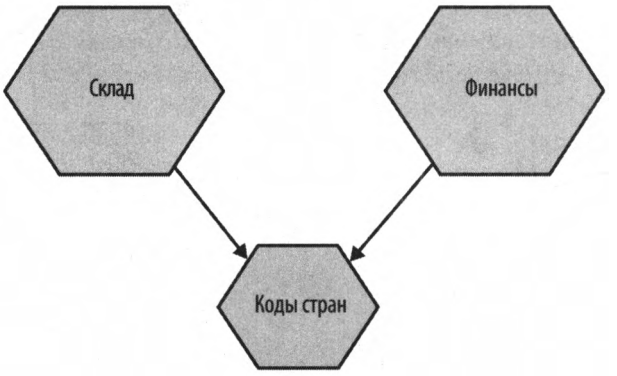


Рис. 4.44. Раздача кодов стран из выделенной службы

Я прокручивал такой сценарий с группами по всему миру, и он имеет тенденцию делить аудиторию пополам. Некоторые люди сразу же думают: "Он сработает!". Однако бóльшая часть группы, как правило, начинает качать головой и говорить что-то вроде: "Он выглядит безумно!". Копая глубже, мы добираемся до сути их беспокойства: работа и потенциальная сложность им кажется значительной, а выгода — небольшой. При этом слово "перебор" возникает совсем нередко!

Поэтому давайте разведаем шаблон чуть-чуть поглубже. Когда я беседую с людьми и пытаюсь понять, почему некоторые относятся к этой идее хорошо, а другие — нет, то в типичной ситуации это сводится к нескольким вещам. Сотрудники, которые работают в среде, где создание и управление микрослужбами находится на низком уровне, этот вариант рассматривают гораздо чаще. С другой стороны, если для создания новой службы, даже такой простой, как эта, требуются дни или даже недели работы, то люди, понятно, будут от создания такой службы отнекиваться.

Бывший мой коллега и соавтор в издательстве O'Reilly Киф Моррис (Kief Morris)[17] рассказал мне о своем опыте в проекте для крупного международного банка, базирующегося в Великобритании, где потребовался почти год на то, чтобы получить одобрение на первый релиз программно-информационного обеспечения. Прежде чем что-либо могло выйти в "прямой эфир", необходимо было провести консультации более чем с 10 группами внутри банка, причем по всем вопросам, начиная с получения подписанных проектов и заканчивая подготовкой машин к работе по развертыванию. К сожалению, такой опыт далеко не редкость в крупных организациях.

В организациях, где развертывание нового софта требует много ручной работы, одобрений и, возможно, даже необходимости приобретения и настройки нового аппаратного обеспечения, стоимость создания служб велика. Следовательно, в такой среде мне следует быть очень избирательным в создании новых служб — они должны приносить большую добавочную ценность, оправдывая дополнительную работу. При этом создание чего-то вроде кодов стран может оказаться неоправданным. С другой стороны, если бы я мог раскручивать "заготовку" службы и пускать ее в производство в течение дня или меньше, а все остальное она делала бы за меня, то я бы с большей вероятностью рассматривал этот вариант как жизнеспособный.

Или даже лучше, служба "Кодов стран" будет отлично подходить для чего-то вроде платформы "функция-как-служба", такой, как облачные функции Azure или AWS Lambda. Более низкая операционная стоимость для функций привлекательна, и они отлично подходят для простых служб, таких, как служба "Кодов стран".

Еще одна упомянутая обеспокоенность заключается в том, что, добавляя службу для кодов стран, мы будем привносить еще одну сетевую зависимость, которая повлияет на задержку. Я считаю, что этот подход не будет хуже, а, может, окажется даже быстрее, чем наличие выделенной базы данных для этой информации. Почему? Дело в том, что, как мы уже установили, в этом наборе данных всего 249 записей. Служба "Кодов стран" сможет легко хранить их в памяти и раздавать напрямую. Наша служба "Кодов стран", скорее всего, будет просто хранить эти записи в коде, без "выпечки" хранилища данных.

Эти данные, конечно, также могут агрессивно кэшироваться на стороне клиента. В конце концов, мы не часто добавляем сюда новые записи! Мы могли бы также подумать об использовании событий для информирования пользователей об изменении этих данных, как показано на рис. 4.45. Когда данные изменяются, заинтересованные потребители предупреждаются с помощью событий и обновляют свои локальные кэши. Я подозреваю, что в этом сценарии традиционный клиентский кэш на основе TTL, вероятно, будет достаточно хорош, учитывая низкую частоту изменений, и я много лет назад с большим эффектом применил аналогичный подход для службы общего назначения "Справочных данных".

---

[17] Киф написал книгу "Инфраструктура как код: управление серверами в облаке (Infrastructure as Code: Managing Servers in the Cloud, Sebastopol: O'Reilly, 2016).

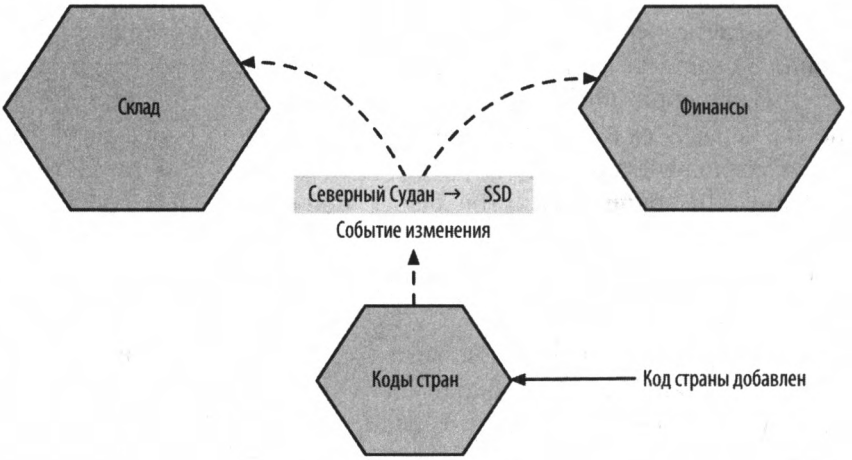


Рис. 4.45. Запуск событий обновления, позволяющих потребителям обновлять локальные кэши

Где его использовать

Я выбрал бы указанный вариант, если бы управлял жизненным циклом этих данных в коде. Например, при желании выставить наружу API для обновления этих данных мне потребовалось бы какое-то место для размещения этого кода, и это целесообразно делать в выделенной микрослужбе. В этой точке у нас есть микрослужба, заключающая в себе машину состояний для этого состояния. Шаблон также имеет смысл, если вы хотите эмитировать события, когда эти данные изменяются, или просто там, где вы хотите обеспечить более удобный контакт, относительно которого можно ставить заглушки для целей тестирования.

Главенствующее затруднение здесь всегда сводится к стоимости создания еще одной микрослужбы. Оправданна ли дополнительная работа, или же один из приведенных ранее других подходов будет более разумным вариантом?

Что бы я сделал?

Я снова надавал вам кучу вариантов. И как же поступить? Наверное, я не могу вечно "сидеть на заборе", так что говорю, как есть. Если считать, что нам не нужно постоянно обеспечивать согласованность кодов стран во всех службах, то я, скорее всего, сохраню эту информацию в совместной библиотеке. Для такого рода данных это, по-видимому, имеет гораздо больший смысл, чем дублирование этих данных в схемах локальных служб, поскольку данные просты по своей природе и невелики по объему (коды стран, размеры одежды и т. п.). Более сложные справочные данные или более крупные их объемы, возможно, подсказали бы мне о том, что следует поместить их в базу данных, локальную для каждой службы.

Если данные должны быть согласованными между службами, я захотел бы создать выделенную службу (или, возможно, раздавать эти данные в рамках более масштабной статической справочной службы).

Я, скорее всего, прибегну к выделенной схеме для такого рода данных только в том случае, если будет сложно оправдать работу по созданию новой службы.

В предыдущих примерах мы рассмотрели несколько рефакторизаций баз данных, которые помогают выделять ваши схемы. Для более детального обсуждения данного вопроса вы, возможно, захотите взглянуть на книгу "Рефакторизация баз данных Скотта Дж. Амблера и Прамод Ж. Садаладжа (Refactoring Databases, Scott J. Ambler, Pramod J. Sadalage, Addison-Wesley).

Транзакции

Разбивая наши базы данных, мы уже коснулись некоторых возникающих проблем. Поддерживать ссылочную целостность становится проблематично, задержка увеличивается, и мы усложняем такие действия, как отчетность. Мы взглянули на различные шаблоны для преодоления некоторых из этих проблем, но одна большая пока остается: как насчет транзакций?

Возможность вносить изменения в базу данных одной транзакцией намного упростит рассуждение о наших системах, а следовательно, облегчит разработку и сопровождение. Мы полагаемся на базу данных в том, что она обеспечивает безопасность и согласованность наших данных, оставляя нам беспокойство о других вещах. Но при разделении данных по базам данных мы теряем выгоду от использования транзакции применительно к изменениям в состоянии атомарным способом.

Прежде чем мы разведаем пути решения этой проблемы, давайте кратко взглянем на то, что дает нам обычная транзакция в базах данных.

ACID-транзакции

В типичной ситуации, когда упоминают транзакции в базах данных, обычно говорят о транзакциях ACID. ACID — это аббревиатура, обозначающая ключевые свойства транзакций в базах данных, которые приводят к созданию системы, на которую мы можем положиться в обеспечении долговечности и согласованности хранения наших данных. ACID означает атомарность (atomicity), согласованность (consistency), изоляцию (isolation) и долговечность (durability), и вот что эти свойства нам дают:

- *Атомарность* — обеспечивает, чтобы все операции, завершаемые в рамках транзакции, либо все завершались успешно, либо все завершались безуспешно. Если какое-либо из изменений, которые мы пытаемся внести, по какой-то причине не срабатывает, то вся операция прерывается, и возникает ситуация, будто никаких изменений никогда не было сделано.
- *Согласованность* — при внесении изменений в базу данных мы обеспечиваем, чтобы она оставалась в допустимом, согласованном состоянии.
- *Изоляция* — позволяет нескольким транзакциям работать одновременно без какого-либо вмешательства. Это достигается путем обеспечения "невидимости" любых промежуточных изменений состояния, произведенных в ходе одной транзакции, для других транзакций.

- *Долговечность* — обеспечивает, чтобы после завершения транзакции мы были уверены, что данные не будут потеряны в случае системного сбоя.

Стоит отметить, что не все базы данных предоставляют транзакции ACID. Все реляционные системы баз данных, которые я когда-либо использовал, их предоставляют, как и многие из новых баз данных NoSQL, таких, как Neo4j. MongoDB в течение многих лет поддерживала транзакции ACID только относительно одного документа, что создает трудности, если вы хотите выполнить атомарное обновление более чем одного документа[18].

Эта книга не предназначена для подробного, глубокого погружения в упомянутые понятия, и ради краткости я, конечно же, упростил некоторые из этих описаний. Для тех из вас, кто хотел бы разведать эти концепции глубже, я рекомендую книгу "Дизайн приложений с интенсивным использованием данных" (Designing Data-Intensive Applications)[19]. В дальнейшем мы будем в основном касаться атомарности. Это не значит, что другие свойства нам не важны, но атомарность тяготеет к тому, чтобы быть первой трудностью, с которой мы сталкиваемся при разложении транзакционных границ.

По-прежнему ACID, но не хватает атомарности?

Я хочу сразу прояснить, что мы по-прежнему можем использовать транзакции в стиле ACID, когда разбиваем базы данных, но объем этих транзакций уменьшается, как и их полезность. Рассмотрим рис. 4.46. Предположим, мы отслеживаем процесс, связанный с подключением нового клиента к нашей системе. Мы подошли к концу процесса, который предусматривает изменение статуса клиента с ОЖИДАЮЩЕГО на ВЕРИФИЦИРОВАННЫЙ. Поскольку регистрация теперь завершена, мы также хотим удалить соответствующую строку из таблицы "Ожидающие регистрации". С одной базой данных это делается в рамках одной транзакции ACID — либо обе новые строки записываются, либо ни одна из них не записывается.

Сравните это с рис. 4.47. Мы делаем точно такие же изменения, но теперь каждое из них вносится в разную базу данных. Это означает, что существует две транзакции, каждая из которых может сработать либо не сработать независимо от другой.

Конечно, мы могли бы задать последовательность этих двух транзакций, удаляя строку из таблицы "Ожидающие регистрации", только если есть возможность изменить строку в таблице "Клиенты". Но нам все равно придется рассуждать о том, что делать, если затем удаление из таблицы "Ожидающие регистрации" не сработало — всю логику нам пришлось бы имплементировать самостоятельно. Обеспечение возможности переупорядочить шаги для учета этих вариантов — действительно полезная идея (к которой мы вернемся, когда займемся разведыванием саг). Но

---

[18] Теперь с поддержкой мультидокументных транзакций ACID, которая была выпущена в рамках Mongo 4.0, ситуация изменилась. Я сам эту особенность Mongo не использовал; я просто знаю, что она существует!

[19] См. Мартин Клеппманн "Дизайн приложений с интенсивным использованием данных" (Martin Kleppmann, Designing Data-Intensive Applications, Sebastopol, O'Reilly Media, Inc., 2017).

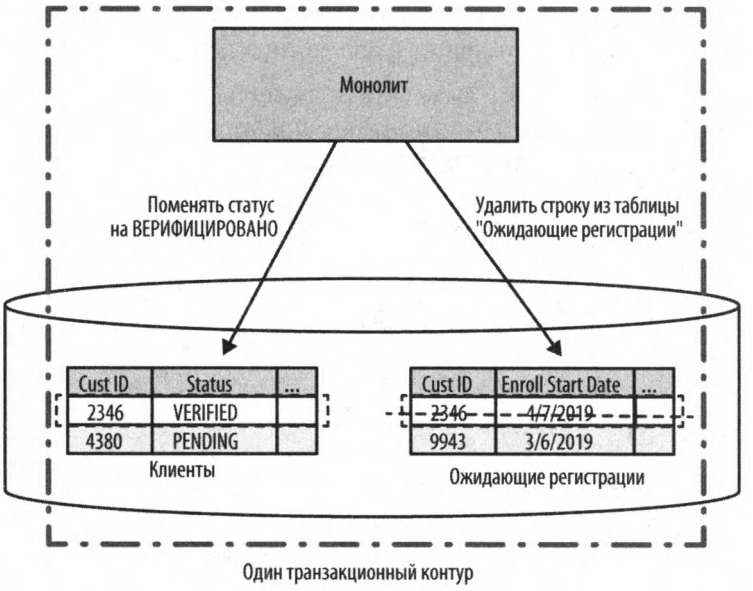


Рис. 4.46. Обновление двух таблиц в области действия одной транзакции ACID

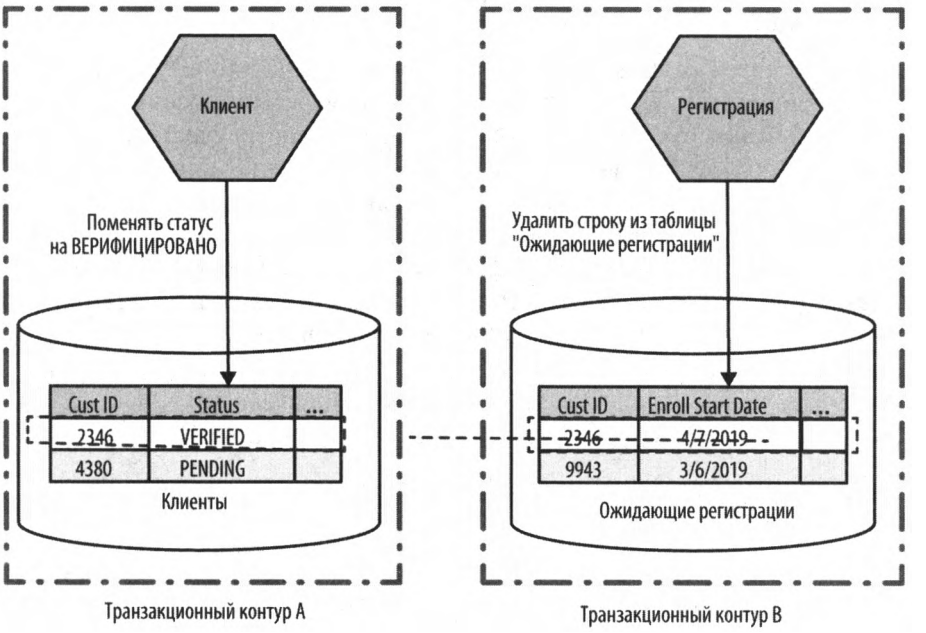


Рис. 4.47. Изменения, вносимые как в "Счет-фактуру", так и в "Заказ", теперь делаются в рамках двух разных транзакций

в принципе, разложив эту операцию на две отдельные транзакции, мы должны признать, что гарантированная атомарность операции в целом потеряна.

Отсутствие атомарности начнет вызывать значительные проблемы, в особенности если мы мигрируем системы, которые ранее полагались на это свойство. Именно в этой точке люди начинают искать другие решения, которые дали бы им возможность рассуждать об изменениях, вносимых в многочисленные службы одновременно. В обычной ситуации первый вариант, который начинают рассматривать, — это распределенные транзакции. Возьмем один из наиболее распространенных алгоритмов имплементации распределенных транзакций — двухфазную фиксацию — как способ разведывания трудностей, ассоциированных с распределенными транзакциями в целом.

Двухфазные фиксации

Алгоритм двухфазной фиксации (иногда сокращенно 2PC от англ. two-phase commit) часто предоставляет нам возможность вносить транзакционные изменения в распределенную систему, где несколько отдельных процессов нуждаются в обновлении в рамках совокупной операции. Я хочу, чтобы вы знали заранее, что двухфазная фиксация имеет ограничения, которые мы в дальнейшем рассмотрим. Распределенные транзакции и, в частности, двухфазные фиксации часто выдвигаются группами, переходящими на архитектуры, основанные на микрослужбах, как способ решения трудностей, с которыми они сталкиваются. Но, как мы увидим, они не решат ваши проблемы и принесут еще больше путаницы в вашу систему.

Указанный алгоритм разбит на две фазы (отсюда и название "двухфазная фиксация"): фаза голосования и фаза фиксации. Во время фазы голосования центральный координатор контактирует со всеми работниками, которые участвуют в транзакции, и запрашивает подтверждение относительно того, можно или нет сделать какое-либо изменение состояния. На рис. 4.48 мы видим два запроса: один на изменение статуса клиента на ВЕРИФИЦИРОВАНО, другой на удаление строки из нашей таблицы "Ожидающие регистрации". Если все работники согласны с тем, что изменение состояния, которое они запрашивают, может произойти, то алгоритм переходит к следующей фазе. Если кто-либо из работников говорит, что изменение произойти не может, возможно, потому, что запрошенное изменение состояния нарушает какое-то локальное условие, то вся операция прерывается.

Важно подчеркнуть, что изменение не вступает в силу сразу после того, как работник сообщает, что он может внести изменения. Вместо этого работник гарантирует, что он сможет сделать это изменение в какой-то момент в будущем. Как работник может дать такую гарантию? Например, на рис. 4.48 работник *A* сказал, что он будет способен изменить состояние строки в таблице "Клиенты" для обновления статуса конкретного клиента до ВЕРИФИЦИРОВАНО. Что делать, если другая операция в какой-то более поздний момент удаляет строку или вносит другое меньшее изменение, которое, тем не менее, означает, что изменение на ВЕРИФИЦИРОВАНО позже является недопустимым? Для того чтобы гарантировать возможность последующего изменения, работнику *A*, вероятно, придется поставить эту запись "на замок", чтобы такое изменение не могло произойти.

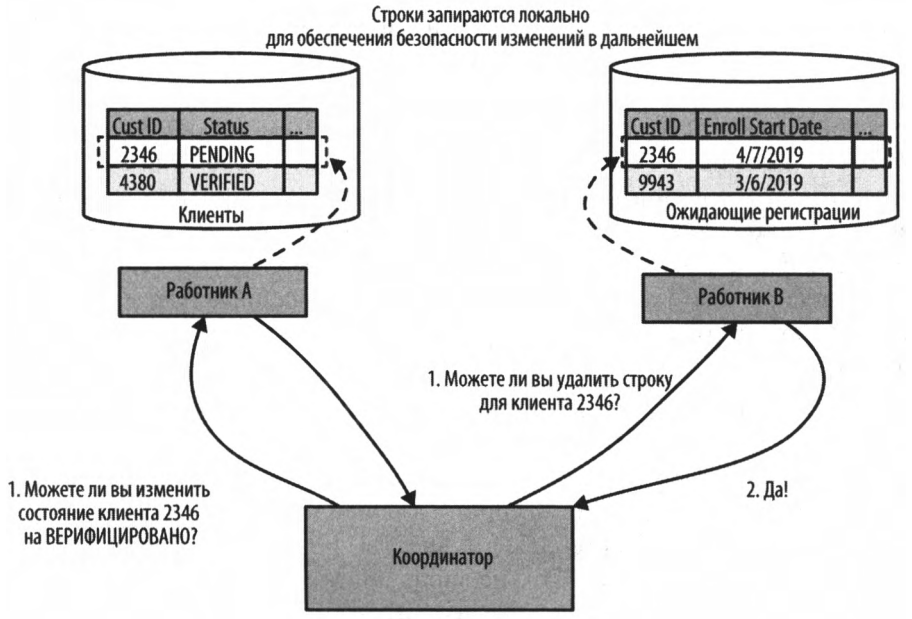


Рис. 4.48. В первой фазе двухфазной фиксации работники голосуют, чтобы решить вопрос выполнения ими нескольких локальных изменений состояния

Если какие-либо работники не проголосовали в пользу фиксации, то сообщение об откате должно быть отправлено всем сторонам для выполнения ими локальной очистки, что позволяет работникам высвободить любые "замки", которые они, возможно, удерживают. Если все работники согласились внести изменения, то мы переходим к фазе фиксации, как показано на рис. 4.49. Здесь изменения вносятся фактически, и высвобождаются ассоциированные "замки".

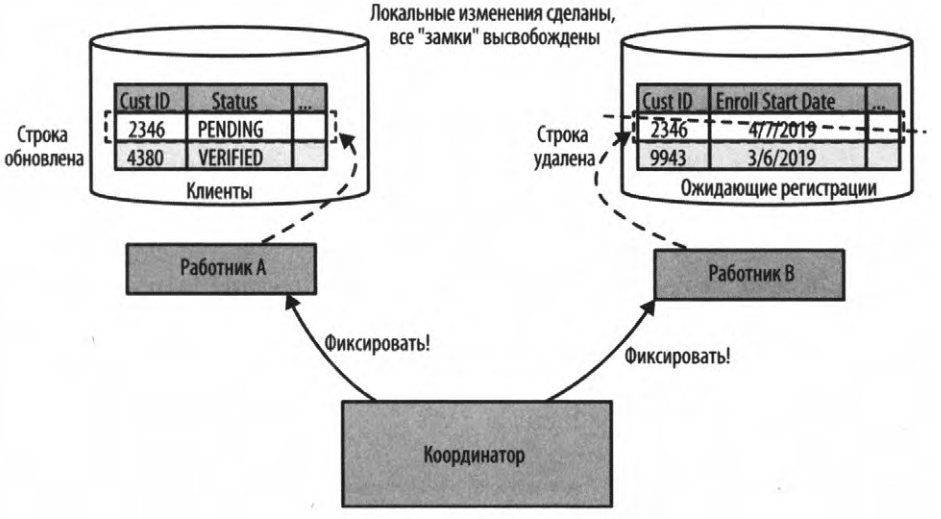


Рис. 4.49. В фазе фиксации двухфазной фиксации изменения применяются фактически

Важно отметить, что в такой системе мы никоим образом не можем гарантировать, что эти фиксации будут происходить в одно и то же время. Координатор должен направить запрос на фиксацию всем участникам, и это сообщение может быть получено и обработано в разное время. Это означает, что мы будем видеть изменения, внесенные в работнике *A*, но еще не видеть изменения в работнике *B*, если обеспечим возможность просматривать состояния этих работников вне координатора транзакции. Чем больше задержка между координатором и чем медленнее работники обрабатывают отклик, тем шире будет это окно несогласованности. Возвращаясь к нашему определению термина ACID, изоляция обеспечивает, чтобы мы не видели промежуточных состояний во время транзакции. Но при двухфазной фиксации мы ее потеряли.

Когда работают двухфазные фиксации, в своей основе они очень часто просто координируют распределенные "замки". Работникам нужно "запереть" доступ к локальным ресурсам для обеспечения возможности фиксации во второй фазе. Управлять "замками" и предотвращать "тупики" в однопроцессной системе — работа не из веселых. Теперь представьте себе трудности координирования "замков" между многочисленными участниками. Это не очень приятно.

С двухфазными фиксациями ассоциирована масса режимов сбоя, которые у нас нет времени разведывать. Возьмем проблему, когда работник голосует за продолжение транзакции, но затем не откликается на запрос о фиксации. Что тогда нам делать? Одни из режимов сбоя будут улажены автоматически, но другие оставят систему в таком состоянии, что придется расшивать вещи вручную.

Чем больше участников и чем значительнее задержка у вас в системе, тем больше затруднений будет иметь двухфазная фиксация. Они будут очевидным способом "вбрызнуть" в вашу систему огромные объемы задержек, в особенности если масштаб запирания на "замок" большой или длительность транзакции велика. Именно по этой причине двухфазные фиксации в типичной ситуации используются только для очень короткоживущих операций. Чем дольше длится операция, тем дольше у вас ресурсы заперты!

Распределенные транзакции — просто скажи "нет"

По всем изложенным ранее причинам я настоятельно рекомендую избегать распределенных транзакций, таких, как двухфазная фиксация, для координации изменений состояния в ваших микрослужбах. Тогда что еще вы можете сделать?

Скажем так, первый вариант — вообще не разбивать данные. Если у вас есть фрагменты состояния, которыми вы хотите управлять по-настоящему атомарным и согласованным способом, и вы не можете решить вопрос, как разумно получить эти характеристики без транзакции в стиле ACID, то оставьте это состояние в одной базе данных и сохраните функциональность, которая управляет этим состоянием в одной службе (или в вашем монолите). Если вы находитесь в процессе выяснения того, где разбить ваш монолит, и определения того, какие декомпозиции являются легкими (или трудными), то вы вполне можете решить, что заниматься разбиением данных, которые в настоящее время управляются в транзакции, просто слишком

сложно прямо сейчас. Поработайте над каким-нибудь другим участком системы и вернитесь к этому вопросу позже.

Но что произойдет, если вам действительно нужно разбить данные, но вы не хотите иметь все тяготы управления распределенными транзакциями? Как выполнять операции в многочисленных службах, но избежать запирания на "замок"? Что, если операция займет минуты, дни или даже месяцы? В подобных случаях мы можем подумать об альтернативном подходе: о сагах.

Саги

В отличие от двухфазной фиксации, *сага* (saga) по своему дизайну представляет собой алгоритм, который координирует многочисленные изменения в состоянии, но позволяет избегать необходимости запирания ресурсов на "замок" в течение длительного периода времени. Мы делаем это путем моделирования шагов, привлекаемых в качестве дискретных действий, которые исполняются независимо. Это сопровождается добавочной выгодой, заставляя нас явно моделировать наши бизнес-процессы, что может дать значительные выгоды.

Стержневая идея, впервые изложенная Гектором Гарсиа-Молиной (Hector Garcia-Molina) и Кеннетом Салемом (Kenneth Salem)[20], отражала трудности, связанные с тем, как лучше всего регулировать операции, которые они называли долгоживущими транзакциями (long lived transaction, LLT). Подобные транзакции могут занимать много времени (минуты, часы или, возможно, даже дни) и в рамках этого процесса требуют, чтобы в базу данных вносились изменения.

Если бы вы непосредственно отображали долгоживущую транзакцию (LLT) в обычную транзакцию базы данных, то одна транзакция охватывала бы весь жизненный цикл LLT. Это привело бы к запиранию нескольких строк или даже полных таблиц на длительные сроки во время выполнения LLT, вызывая значительные затруднения, если другие процессы пытаются прочитать или модифицировать запертые ресурсы.

Вместо этого авторы статьи предлагают разбить LLT на последовательность транзакций, каждая из которых регулируется независимо. Идея заключается в том, что продолжительность каждой "под-транзакции" будет меньшей и будет модифицирована только часть данных, затронутых всей LLT-транзакцией. Как результат, в опорной базе данных окажется гораздо меньше конкуренции, т. к. объем и продолжительность "замков" значительно уменьшаются.

Первоначально саги предусматривались в качестве механизма, помогающего LLT-транзакциям действовать в отношении одной базы данных, однако указанная модель работает так же хорошо для координирования изменений в многочисленных службах. Мы можем разбить один бизнес-процесс на множество вызовов сотрудничающих служб в рамках одной саги.

---

[20] См. Гектор Гарсия-Молина и Кеннет Салем "Саги" (Sagas, Hector Garcia-Molina, Kenneth Salem, в ACM Sigmod Record 16, № 3 (1987): 249–259.

Прежде чем мы пойдем дальше, вы должны понять, что сага не дает нам атомарности в терминах ACID, к которой мы привыкли с обычной транзакцией базы данных. Поскольку мы разбиваем LLT на отдельные транзакции, у нас нет атомарности на уровне самой саги. У нас есть атомарность для каждой "под-транзакции" внутри LLT, поскольку каждая из них может относиться к ACID-транзакционному изменению, если это необходимо. Но вот что сага нам дает, так это достаточно информации для выяснения состояния, в котором она находится. Ответ на вопрос, как обращаться с последствиями этого, находится полностью в наших руках.

Давайте взглянем на простой поток исполнения заказа, изображенный на рис. 4.50, который подходит для дальнейшего разведывания саг в контексте архитектуры на основе микрослужб.

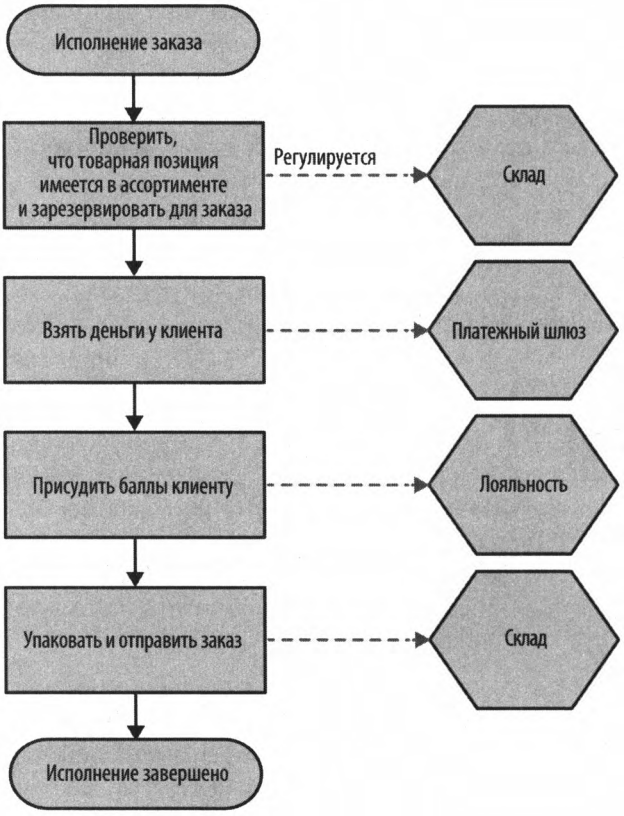


Рис. 4.50. Пример потока исполнения заказа наряду со службами, ответственными за проведение операции

Здесь процесс исполнения заказа представлен как одна сага, причем каждый шаг в этом потоке представляет собой операцию, которая выполняется другой службой. В каждой службе любое изменение состояния регулируется в рамках локальной ACID-транзакции. Например, когда мы проверяем и резервируем запасы с помощью службы "Склад", эта служба, возможно, создаст строку внутри своей локальной таблицы "Резервирование", регистрирующую резервирование, и это изменение будет регулироваться внутри обычной транзакции.

Режимы сбоя саги

Когда сага разбивается на отдельные транзакции, нам нужно подумать о том, как работать со сбоем или, конкретнее, как восстанавливаться, когда происходит сбой. В оригинальной статье "Сага" описаны два типа: обратное и прямое восстановление.

Обратное восстановление предусматривает отмену сбоя с возвратом назад и последующую очистку — откат. Чтобы это работало как надо, необходимо определить компенсирующие действия, которые позволят нам отменять ранее совершенные транзакции. Прямое восстановление позволяет нам восстанавливаться из точки, где произошел сбой, и продолжать обработку. А для того чтобы и это работало как надо, необходимо, чтобы у нас была возможность делать повторные попытки транзакций, из чего, в свою очередь, вытекает, что наша система поддерживает постоянство достаточного объема информации для осуществления повторной попытки.

В зависимости от природы моделируемого бизнес-процесса вы можете предусмотреть, чтобы любой режим сбоя запускал обратное восстановление, прямое восстановление или, возможно, их смесь.

Откаты в саге

В случае ACID-транзакции откат происходит перед фиксацией. После отката ситуация такова, будто ничего и не произошло: изменение, которое мы пытались внести, не состоялось. Однако с сагой у нас вовлечены многочисленные транзакции, и некоторые из них, возможно, уже зафиксированы до того, как мы решим откатить всю операцию. Как же откатить транзакции после того, как они уже были зафиксированы?

Давайте вернемся к нашему примеру с обработкой заказа, как показано на рис. 4.50. Возьмем потенциальный режим сбоя. Мы дошли до попытки упаковать товарную позицию и обнаружили, что указанная товарная позиция на складе не найдена, как показано на рис. 4.51. Наша система думает, что товарная позиция существует, но ее просто нет на полке!

Теперь давайте допустим, что мы решили просто откатить весь заказ, вместо того чтобы дать клиенту возможность разместить товарную позицию в своем заказе на отсутствующий товар. Проблема в том, что мы уже приняли оплату и присудили за заказ баллы за лояльность.

Если бы все шаги выполнялись в одной транзакции, то простой откат все это очистил бы. Тем не менее каждый шаг в процессе исполнения заказа регулировался другим вызовом службы, каждый из которых оперировал в другом транзакционном диапазоне. Простого "отката" для всей операции не существует.

Вместо этого, если вы хотите имплементировать откат, то вам нужно имплементировать компенсирующую транзакцию. Компенсирующая транзакция — это операция, которая отменяет ранее зафиксированную транзакцию. Для отката процесса исполнения заказа мы запускаем компенсирующую транзакцию для каждого шага нашей саги, которая уже была зафиксирована, как показано на рис. 4.52.

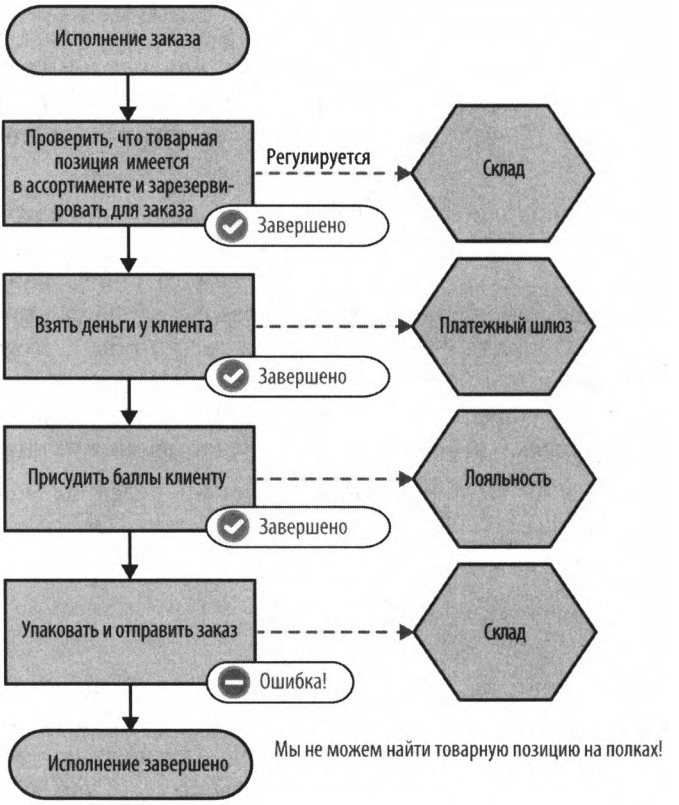


Рис. 4.51. Мы попытались упаковать нашу товарную позицию, но не можем найти ее на складе

Стоит отметить тот факт, что компенсирующие транзакции не обладают точно таким же поведением, как и обычный откат базы данных. Откат базы данных происходит перед фиксацией; и после отката возникает состояние, будто бы транзакция никогда не происходила. В этой ситуации, конечно, такие транзакции действительно имели место. Мы создаем новую транзакцию, которая отменяет изменения, внесенные исходной транзакцией, но мы не можем откатить назад время и сделать так, как будто исходная транзакция не произошла.

Поскольку не всегда есть возможность откатывать транзакцию назад "чисто", мы говорим, что компенсирующие транзакции являются семантическими откатами. Мы не можем все всегда подчищать, но мы делаем достаточно для контекста нашей саги. Например, один из наших шагов, возможно, предусматривал отправку электронной почты клиенту с сообщением о том, что его заказ уже в пути. Если мы решим откатить его назад, то вы не сможете вернуть отправленное электронное письмо![21] Вместо этого ваша компенсирующая транзакция может активировать отправку клиенту второго электронного письма с сообщением о том, что возникла проблема с заказом и он был отменен.

[21] Вы действительно не сможете. Я пытался!

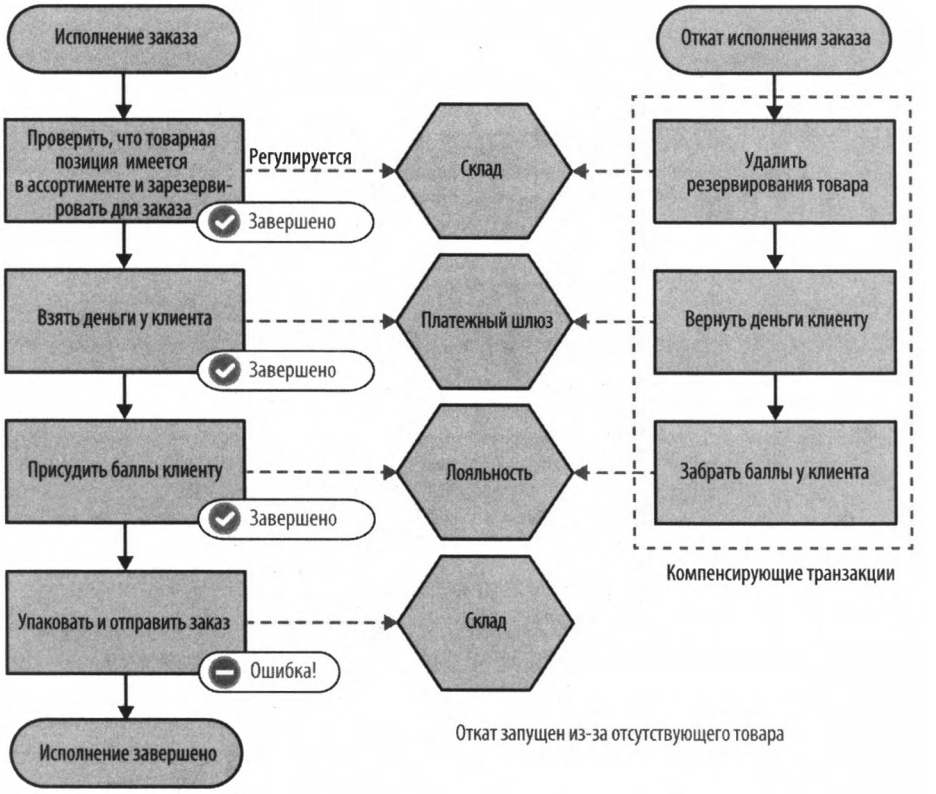


Рис. 4.52. Запуск отката всей саги

Абсолютно уместно, чтобы в системе обеспечивалось постоянство информации, связанной с откатом саги. На самом деле эта информация может оказаться очень важной. Возможно, вы захотите по целому ряду причин оставить запись в службе "Заказы" по прерванному заказу, а также информацию о том, что произошло.

Переупорядочивание шагов для уменьшения откатов

На рис. 4.52 мы могли бы сделать наши вероятные сценарии отката несколько проще, переупорядочив шаги. Пример несложного изменения — присуждение баллов только тогда, когда заказ был фактически отправлен, как показано на рис. 4.53. И нам не придется беспокоиться о том, чтобы этот этап был откатан назад, если у нас возникнут проблемы при попытке упаковать и отправить заказ. Иногда вы можете упростить операции отката, просто подрегулировав то, как этот процесс выполняется. Проталкивая вперед те шаги, которые с наибольшей вероятностью не сработают, и приводя процесс к неуспеху раньше, вы избегаете необходимости запускать более поздние компенсирующие транзакции, поскольку эти шаги даже вообще не были инициированы.

Если разместить подобные изменения, то ваша жизнь станет намного проще, даже отпадет необходимость создавать компенсирующие транзакции для некоторых шагов. Это будет особенно важно, если имплементация компенсирующей транзакции

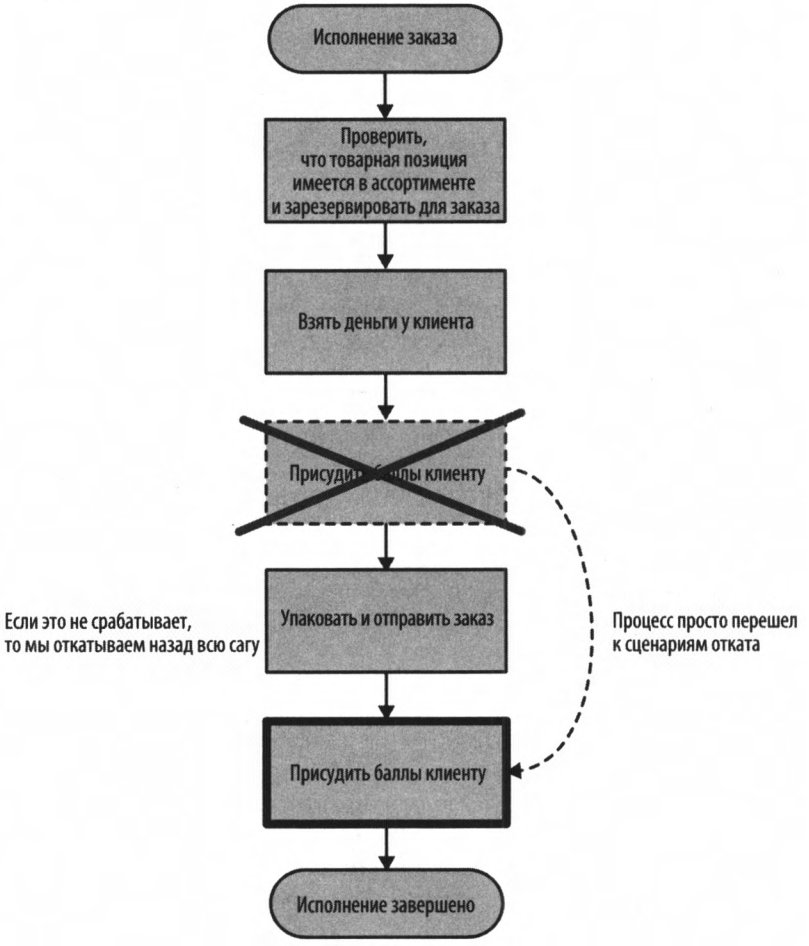


Рис. 4.53. Перенос шагов на более поздние участки в саге уменьшает объемы отката назад в случае сбоя

затруднена. Вы будете в состоянии перенести шаг на более поздний участок в указанном процессе, где никогда не возникнет потребность откатить его назад.

Смешивание ситуаций сбоя назад и сбоя вперед

Вполне уместно сочетать режимы восстановления после сбоя. Некоторые сбои могут потребовать отката назад; другие могут быть сбоем вперед. Для обработки заказа, например, после того как мы взяли деньги у клиента и товарная позиция была упакована, остается только отправить пакет. Если по какой-то причине мы не можем отправить пакет (возможно, у нашей фирмы по доставке нет места в фургонах, чтобы взять заказ сегодня), откат всего заказа назад будет выглядеть странно. Вместо этого мы, вероятно, просто повторим попытку отправки, и если она сработает, то она потребует вмешательства человека для улаживания ситуации.

Имплементация саг

До сих пор мы рассматривали логическую модель того, как саги работают, но нам нужно пойти немного глубже и проэкзаменовать способы имплементации самой саги. Мы можем рассмотреть два стиля имплементации саг. Оркестрированные саги теснее следуют за оригинальным пространством решения и в первую очередь опираются на централизованную координацию и отслеживание. Их можно сравнить с хореографированными сагами, которые избегают необходимости централизованной координации в пользу более слабо сопряженной модели, но усложняют отслеживание хода выполнения саги.

Оркестрированные саги

Оркестрированные саги (orchestrated saga) используют центрального координатора (то, что мы далее будем называть оркестровщиком) для определения порядка исполнения и активации любого необходимого компенсирующего действия. Вы можете думать об оркестрированных сагах как о командно-контрольном подходе: центральный оркестровщик контролирует, что происходит и когда, и в результате возникает хорошая степень видимости того, что происходит с любой данной сагой.

Взяв процесс исполнения заказа, показанный на рис. 4.50, давайте посмотрим, как этот процесс центральной координации будет работать в виде набора сотрудничающих служб, как показано на рис. 4.54.

Здесь наш центральный "Обработчик заказов", играя роль оркестровщика, координирует процесс исполнения. Он знает, какие службы необходимы для выполнения

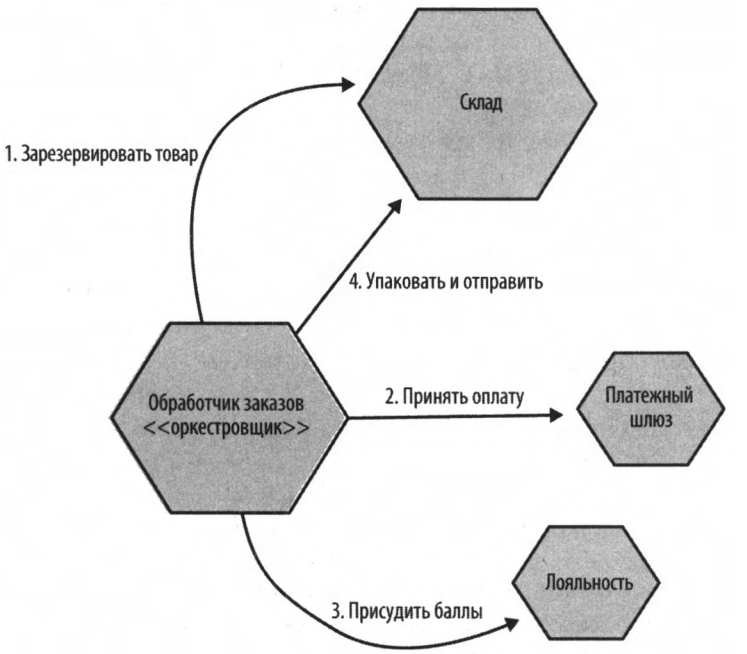


Рис. 4.54. Пример того, как оркестрированная сага используется для имплементации нашего процесса исполнения заказа

операции, и решает, когда следует вызывать эти службы. Если вызовы не срабатывают, то он решает, что делать в результате. Эти оркестрированные обработчики, как правило, интенсивно используют вызовы "запрос-отклик" между службами: "Обработчик заказов" отправляет запрос в службы (например, "Платежный шлюз") и ожидает отклика, сообщающего ему об успешности или неуспешности запроса и предоставляющего результаты запроса.

Чрезвычайно полезно, чтобы наш бизнес-процесс был однозначно смоделирован внутри "Обработчика заказов". Это позволяет взглянуть на одно место в нашей системе и понять, как этот процесс предположительно должен работать. В результате облегчается интеграция новых сотрудников и обмен знаниями о стержневых частях системы.

Тем не менее здесь следует учесть несколько недостатков. Прежде всего, по своей природе этот подход является несколько сопряженным. Наш "Обработчик заказов" должен знать обо всех ассоциированных службах, что приводит к более высокой степени доменной сопряженности (см. *главу 1*). Хотя она не так уж и плоха, мы все равно хотели бы свести доменную сопряженность к минимуму, если это возможно. Здесь нашему "Обработчику заказов" нужно знать и контролировать так много вещей, что этой формы сопряженности трудно избежать.

Другая, более тонкая, трудность заключается в том, что логика, которая в иных случаях должна быть "втиснута" в службы, вместо этого может начать поглощаться оркестровщиком. Если подобное начнет происходить, то вы обнаружите, что ваши службы становятся "анемичными", с малым собственным поведением, просто принимая заказы от оркестровщиков, таких, как "Обработчик заказов". Важно, чтобы вы по-прежнему рассматривали службы, составляющие эти оркестрированные потоки, как сущности, имеющие свое собственное локальное состояние и поведение. Они отвечают за свои собственные локальные машины состояний.

Если у логики есть место, где она может быть централизована, то она станет централизованной!

Один из способов избежать чрезмерной централизации с оркестрированными потоками состоит в обеспечении наличия разных служб, выполняющих роль оркестровщика для разных потоков. У вас может быть служба "Обработка заказов", которая улаживает размещение заказа, служба "Возвраты" для обеспечения процесса возврата товаров и уплаченных денег, служба "Прием товаров", которая улаживает поступление на склад новых товаров и расстановку их на полках, и т. д. Что-то вроде нашей службы "Склад" может использоваться всеми этими оркестровщиками. Такая модель упрощает поддержание функциональности в самой службе "Склад", обеспечивая функциональность во всех этих потоках повторно.

Инструменты моделирования бизнес-процессов?

Инструменты моделирования бизнес-процессов (business process modeling, BPM) существуют уже много лет. По большому счету, они предназначены для обеспечения неразработчиков возможностью определять потоки бизнес-процессов, часто через ви-

зуальные инструменты перетаскивания. Идея заключается в том, что разработчики создают строительные блоки этих процессов, а затем неразработчики связывают эти строительные блоки вместе в более крупные процессные потоки. Применение таких инструментов, похоже, очень хорошо выстраивается как способ имплементации оркестрированных саг, и, действительно, процессная оркестровка — это в значительной степени главный вариант работы с инструментами BPM (или, наоборот, использование инструментов BPM приводит к тому, что вам требуется оркестровка).

С течением времени инструменты BPM мне, в конце концов, сильно перестали нравиться. Главная причина в том, что центральная высокомерная идея — о том, что неразработчики будут определять бизнес-процесс, — на моем опыте почти никогда не была верной. Инструментарий, предназначенный для неразработчиков, в конечном итоге используется разработчиками, и у них возникает масса затруднений. Они часто требуют применения GUI-интерфейсов для изменения потоков, потоки, которые они создают, с трудом (или вообще не) поддаются версионному контролю, сами потоки оказываются спланированы без учета тестирования и многое другое.

Если ваши разработчики собираются имплементировать ваши бизнес-процессы, то пускай они задействуют инструментарий, который они знают и понимают и который пригоден для их трудовых потоков. В общем случае, это означает просто дать им возможность имплементации этих вещей непосредственно в коде! Если вам нужна наглядность того, как был имплементирован бизнес-процесс или как он работает, то гораздо проще спроецировать визуальное представление трудового потока из кода, чем описывать с помощью визуального представления, как должен работать ваш код.

Предпринимаются меры по созданию более удобных для разработчиков инструментов BPM. Отзывы об этих инструментах от разработчиков различны, но в ряде случаев они хорошо себя проявили, и приятно видеть, что люди пытаются усовершенствовать эти каркасы. Если вы чувствуете необходимость разведать эти инструменты дальше, взгляните на Camunda[22] и Zeebe[23], оба из которых являются оркестровочными каркасами с открытым исходным кодом, предназначенными для разработчиков микрослужб.

Хореографированные саги

Хореографированные саги (choreographed saga) призваны распределять ответственность за функционирование саги среди нескольких сотрудничающих служб. Если оркестровка является командно-контрольной, то хореографированные саги представляют собой архитектуру по принципу "доверяй, но проверяй". Как мы увидим в нашем примере на рис. 4.55, хореографированные саги интенсивно используют события для сотрудничества между службами.

Здесь происходит довольно много, и поэтому стоит разведать тему подробнее. Прежде всего, эти службы реагируют на получаемые ими события. Концептуально события транслируются в системе, и заинтересованные стороны могут их получать. Вы не отправляете события в некую службу; вы просто их запускаете, и службы, которые заинтересованы в этих событиях, получают их и действуют соответствующим образом. В нашем примере, когда служба "Склад" получает первое событие "Заказ размещен", она знает свою работу, резервируя соответствующий товар и запуская событие, как только резервирование будет сделано. Если товар не может

[22] См. **https://camunda.com/**.

[23] См. **https://zeebe.io/**.

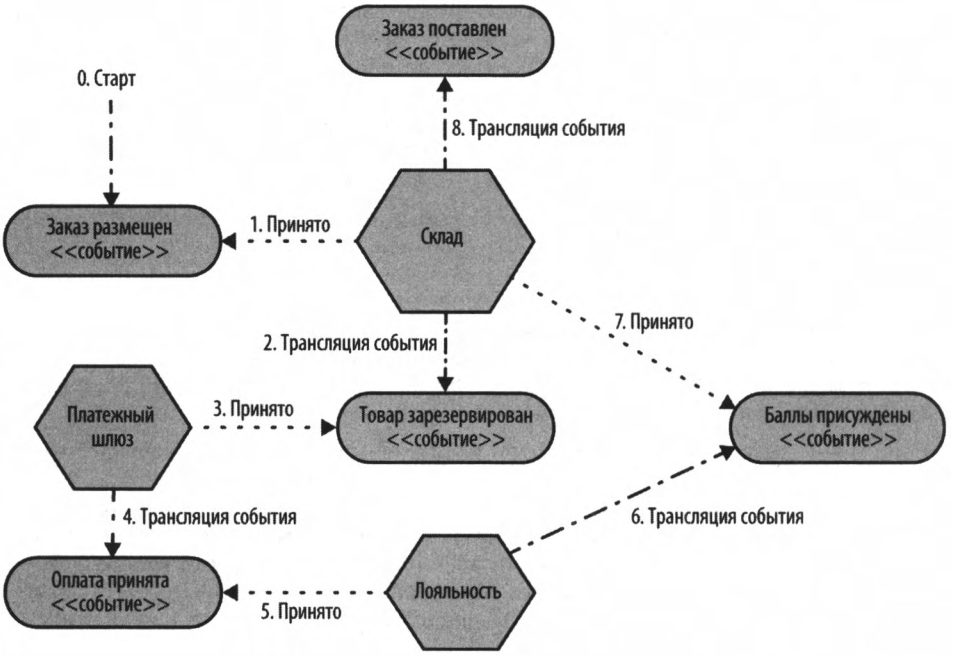


Рис. 4.55. Пример хореографированной саги для имплементации исполнения заказа

быть получен, то "Складу" потребуется вызвать соответствующее событие (возможно, событие "Нехватка товара"), что может привести к прерыванию заказа.

Как правило, для управления надежной трансляцией и доставкой событий вы задействуете своего рода брокера сообщений. Существует возможность, что несколько служб будут реагировать на одно и то же событие, и именно там вы будете использовать тему. Стороны, заинтересованные в определенном типе событий, будут подписываться на определенную тему, не беспокоясь о том, откуда эти события пришли, и брокер обеспечивает долговечность темы и то, что события на ней успешно доставляются подписчикам. Например, у нас может быть служба "Рекомендации", которая также прослушивает события "Заказ размещен" и использует его для построения базы данных вариантов музыки, которые вам могут понравиться.

В приведенной ранее архитектуре ни одна служба не знает ни о какой другой службе. Им нужно только знать, что делать, когда получено определенное событие. По сути, это приводит к гораздо менее сопряженной архитектуре. Поскольку имплементация процесса здесь разлагается и распределяется между четырьмя службами, мы также избегаем опасений по поводу централизации логики (если у вас нет места, где логика может быть централизована, то она не будет централизована!).

Оборотная сторона всего этого заключается в том, что теперь сложнее выяснить, что происходит. С оркестровкой процесс был явно смоделирован в нашем оркестровщике. Теперь, с этой архитектурой, как она представлена, непонятно как выстроить ментальную модель того, каким предположительно должен быть процесс? Вам нужно будет смотреть на поведение каждой службы в изоляции и воссоздавать

эту картину в своей голове — далеко не прямолинейный процесс, даже с таким простым бизнес-процессом, как в нашем примере.

Уже достаточно плохо то, что не хватает четкого представления нашего бизнес-процесса, но у нас также нет способа узнать, в каком состоянии находится сага, что также лишает нас возможности прикрепить компенсирующие действия, когда это необходимо. Мы можем вложить некую ответственность за выполнение компенсационных действий в отдельные службы, но в принципе нам нужен способ узнать, в каком состоянии находится сага для некоторых вариантов восстановления. Отсутствие центрального места, где выполняется опрос о статусе саги, составляет большую проблему. Оркестровка позволяет устранить эту проблему, но как же решить ее здесь?

Один из самых простых способов сделать это — взять проекцию о состоянии саги из существующей системы, потребляя эмитируемые события. Если мы сгенерируем уникальный ИД для саги, то сможем поместить его во все события, которые эмитируются в рамках этой саги, — это называется ИД корреляции. Тогда мы могли бы иметь службу, работа которой состоит в том, чтобы просто вакуумировать все эти события и представлять проекцию о том, в каком состоянии находится каждый заказ, и, возможно, программно выполнять действия по урегулированию трудностей в рамках процесса его исполнения, если другие службы не могут это сделать сами.

Смешивание стилей

Хотя может показаться, что оркестрированные и хореографированные саги диаметрально противоположны по виду имплементации, вы можете легко подумать о смешивании и сочетании указанных моделей. В вашей системе могут иметься некие бизнес-процессы, которые естественнее вписываются в ту или иную модель. Вы также можете иметь единую сагу, которая сочетает смесь стилей. В варианте с исполнением заказов, например, внутри контура службы "Склад", во время управления упаковкой и отправкой пакета мы можем использовать оркестрированный поток, даже если исходный запрос был сделан в рамках более крупной хореографированной саги[24].

Если вы все-таки решите смешать стили, то важно, чтобы у вас по-прежнему имелся четкий способ выяснить, что произошло в рамках саги. Без этого понять режимы сбоя становится сложно, и выполнить восстановление после сбоя трудно.

Что использовать: хореографию или оркестровку?

Имплементация хореографированных саг приносит с собой идеи, которые, возможно, не знакомы вам и вашей группе. Они в типичной ситуации исходят из интенсивного использования событийно-обусловленного сотрудничества, которое не

[24] Это выходит за рамки данной книги, но Гектор Гарсия-Молина и Кеннет Салем продолжили разведывать вопрос о том, как многочисленные саги могут "вкладываться" для имплементации более сложных процессов. Для знакомства с этой темой поближе см. Гектор Гарсия-Молина и др. "Моделирование длительных операций как вложенных саг" (Modeling Long-Running Activities as Nested Sagas, Hector Garcia-Molina et al, Data Engineering 14, No.1, March 1991: 14–18).

очень широко известно. Однако, по моему опыту, лишняя сложность, связанная с отслеживанием хода выполнения саги, почти всегда перевешивается выгодами, ассоциированными с наличием более слабо сопряженной архитектуры.

Отступая от моих личных вкусов, дам общий совет в отношении оркестровки против хореографии: я очень расслаблен в использовании оркестрированных саг, когда одна группа владеет имплементацией всей саги. В такой ситуации гораздо легче управлять в пределах границ группы с более сопряженной архитектурой. Если у вас задействовано несколько групп, то я предпочитаю более декомпозированную хореографированную сагу, поскольку легче распределить ответственность за имплементацию саги между группами, а более слабо сопряженная архитектура позволяет этим группам работать изолированнее.

Саги против распределенных транзакций

Надеюсь, я уже разложил по полочкам, что распределенные транзакции сопровождаются некоторыми значительными трудностями, и я их стараюсь избегать за пределами некоторых очень специфических ситуаций. Пэт Хелланд (Pat Helland), пионер в области распределенных систем, выделяет фундаментальные трудности с имплементацией распределенных транзакций для тех видов приложений, которые мы создаем сегодня[25]:

> "В большинстве систем распределенных транзакций сбой одного узла приводит к остановке фиксации транзакции. Это, в свою очередь, приводит к тому, что приложение “заклинивается”. В таких системах чем больше она становится, тем вероятнее, что система “ляжет”. В пилотировании самолетом, который нуждается в работе всех своих двигателей, добавление двигателя уменьшает годность самолета".
>
> – *Пэт Хелланд,* Жизнь за пределами распределенных транзакций

По моему опыту, явное моделирование бизнес-процессов как саги позволяет избегать многих трудностей распределенных транзакций, но в то же время имеет добавочную выгоду, делая то, что в иных случаях было бы внутренне смоделированными процессами, гораздо более явными и очевидными для ваших разработчиков. Превращение стержневых бизнес-процессов вашей системы в первоклассную концепцию будет иметь массу выгод.

Более полное обсуждение имплементации оркестровки и хореографии, наряду с различными деталями имплементации, выходит за рамки этой книги. Данная тема рассматривается в *главе 4* книги "Создание микросервисов", но я также рекомендую книгу "Шаблоны интеграции предприятия" для глубокого погружения во многие аспекты указанной темы[26].

---

[25] См. Пэт Хелланд "Жизнь за пределами распределенных транзакций" (Life Beyond Distributed Transactions, Pat Helland, acmqueue 14, No. 5).

[26] Саги не упоминаются явно ни в одной из книг, но оркестровка и хореография освещены. Хотя я не могу говорить об опыте авторов книги "Шаблоны интеграции предприятия", лично я не был осведомлен о сагах, когда писал книгу "Создание микросервисов".

Резюме

Мы выполняем декомпозицию нашей системы, находя стыки, вдоль которых возникают контуры служб, и этот подход является поступательным. Научившись находить эти стыки и, в первую очередь, работая над снижением стоимости выделения служб, мы можем продолжать выращивать и эволюционировать наши системы для удовлетворения любых требований, возникающих по пути. Как вы видите, некоторые части этой работы бывают кропотливыми и вызывают значительные трудности, которые нам нужно будет преодолеть. Но тот факт, что это можно сделать поступательно, означает, что нет необходимости бояться такой работы.

При выделении наших служб мы также добавили несколько новых проблем. В следующей главе мы посмотрим на разнообразные трудности, которые появятся по мере того, как вы будете делить свой монолит. Но не волнуйтесь, я также дам вам массу идей, которые помогут справляться с этими проблемами по мере их возникновения.

Болезни роста

Приняв на вооружение архитектуру на основе микрослужб, вы обязательно столкнетесь на этом пути с трудностями. Мы уже вскользь рассматривали некоторые из этих трудностей, но я хочу разведать их дальше, для того чтобы дать вам одно предостережение.

В этой главе я надеюсь дать вам предостаточно информации о тех типах проблем, с которыми вы столкнетесь. Я не смогу решить их все в этой книге, и многие проблемы, которые здесь описаны, более детально изложены в книге "Создание микросервисов", во многом написанной с учетом этих трудностей.

Я также хочу дать рекомендации в отношении того, какие сигналы искать, чтобы помочь вам опознавать моменты, когда эти вопросы будут нуждаться в решении, а также дать указание на то, где на вашем пути подобные вопросы, скорее всего, будут возникать.

Чем больше служб, тем больше боли

Момент, когда именно возникнут проблемы с архитектурой на основе микрослужб, зависит от массы факторов. Сложность взаимодействия служб, размер организации, число служб, выбор технологий, задержки и требования к времени безотказной работы — это лишь подмножество сил, которые приносят боль, страдания, волнение и стресс. Это означает, что трудно сказать, когда вы столкнетесь с данными проблемами или столкнетесь ли вы с ними вообще.

В целом, однако, я понял, что разновидности проблем, которые возникают в компании с десятью службами, как правило, сильно отличаются от тех, которые наблюдаются в компании с сотнями служб. Число служб, по-видимому, является столь же хорошей мерой, как и любая другая, для индикации того, когда, скорее всего, проявятся определенные проблемы. Здесь я должен отметить, что, говоря о числе служб, если не указано иное, я имею в виду разные логические службы. После развертывания этих служб в производстве они затем могут быть развернуты в виде многочисленных экземпляров служб.

Не думайте о принятии на вооружение микрослужб, как о щелчке выключателя, думайте об этом, как о повороте наборного диска. Когда вы поворачиваете этот диск и создаете больше служб, то вы, надо надеяться, будете иметь больше возможностей получить что-то нужное из микрослужб. Но, по мере того как вы поворачиваете этот диск, вы по ходу попадаете в разные болевые точки. И, по мере того

как вы это делаете, вам нужно отыскивать способы разрешения возникающих проблем, которые нередко требуют новых способов мышления, новых навыков, других методов или, возможно, новых технологий.

На рис. 5.1 условно показаны "болевые точки", которые мы рассмотрим далее в этой главе, основываясь на том, где в вашем "выращивании" служб эти трудности, скорее всего, возникнут. Приведенный анализ далек от научного и в значительной степени основан на разрозненном опыте, но я по-прежнему считаю, что он полезен в качестве общего представления.

Рис. 5.1. Обобщенная схема проявления некоторых из "болевых точек"

Я не говорю, что вы обязательно получите все указанные проблемы на эти участках времени или вообще их увидите. Сюда вовлечены определенные переменные, которые такая простая диаграмма не способна передать по-настоящему. Один из факторов, в частности, который может измениться, когда эти трудности нанесут удар, — вопрос, насколько сопряженной является ваша архитектура. При более сопряженной архитектуре трудности, связанные с робастностью, тестированием, трассировкой и т. п., проявятся раньше. Я надеюсь пролить всего лишь немного света на потенциальные ловушки, которые могут встретиться.

Однако имейте в виду, что вы должны использовать приведенную схему в качестве общего индикатора. Вам нужно постараться выстроить механизмы обратной связи, которые помогут отыскивать некоторые из потенциальных индикаторов, которые я здесь описываю.

Теперь, когда я в полном объеме высказал формальное предупреждение об ограничениях схемы, изображенной на рис. 5.1, давайте рассмотрим каждый из этих вопросов немного подробнее. Я намерен дать вам несколько советов о том, какие факторы выведут эти проблемы на первый план, понимание того, как эти трудности повлияют на вас, и некоторые советы по устранению трудностей по мере их возникновения.

Владение кодом в широком масштабе

По мере того как все больше и больше разработчиков трудятся над вашей архитектурой на основе микрослужб, вы доберетесь до того места, где, возможно, пересмотрите управление владением кодом.

С точки зрения универсального владения кодом Мартин Фаулер ранее выделил разные типы владения[1], и я нахожу, что, вообще говоря, они работают и в контексте строительства микрослужб. Здесь в первую очередь мы рассматриваем "владение" с точки зрения внесения изменений в код, а не с точки зрения того, кто управляет развертыванием, поддержкой на первой линии и т. д. Прежде чем мы поговорим о типах проблем, которые возникают, давайте сначала взглянем на концепции, очерченные Мартином, и поместим их в контекст архитектуры на основе микрослужб:

- *Сильная степень владения кодом* — все службы имеют владельцев. Если кто-то за пределами этой группы владения хочет внести изменения, то он должен представить данное изменение владельцам, которые решают, разрешено ли это делать или нет. Использование запроса на включение внесенных изменений для людей вне группы владения — один из примеров того, как это может регулироваться.
- *Слабая степень владения кодом* — большинство служб, если не все, кому-то принадлежат, но любой человек может модифицировать свои модули непосредственно, не прибегая к запросам на включение внесенных изменений. Система управления версиями практически организована так, чтобы все же позволять любому изменять что угодно, но при этом ожидается, что если вы измените чью-то службу, то вы сначала с ним поговорите.
- *Коллективное владение кодом* — никто ничем не владеет, и каждый может изменить все, что захочет.

Как эта проблема проявляется?

По мере роста числа служб и разработчиков у вас возникают проблемы с коллективным владением. Для того чтобы коллективное владение работало, необходим достаточно хорошо взаимосвязанный коллектив. Такая взаимосвязь позволяет ему иметь общепринятое совместное понимание того, как выглядит хорошее изменение и в каком направлении вы хотите воспользоваться конкретной службой с технической точки зрения.

Отложив в сторону отдельные случаи, я убедился, что в широком масштабе коллективное владение кодом было для архитектуры на основе микрослужб катастрофическим. Одна финансово-технологическая компания, с представителями которой я беседовал, поделилась историями о небольшой группе, переживавшей быстрый рост, перейдя с 30–40 разработчиков к более чем 100, но без назначения каких-либо

[1] См. **http://bit.ly/2n5pSAf**.

обязанностей разным частям системы или какой-либо концепции владения, кроме того, что "народ знает, что является правильным".

В результате по мере эволюции системной архитектуры возникло размытое ее видение и ужасно запутанный "распределенный монолит". Один из разработчиков назвал свою архитектуру "дуршлачной архитектурой"[2], потому что она была испещрена дырами — люди просто-напросто "пробивали новую дыру", когда им заблагорассудится, выставляя данные наружу или просто делая много двухточечных вызовов. Реальность такова, что такие трудности легче уладить с помощью монолитной системы, но гораздо сложнее с помощью распределенной — стоимость распутывания распределенного монолита намного выше.

Когда эта проблема возникает?

Для многих групп, начиная с малых, модель коллективного владения кодом имеет смысл. При небольшом числе расположенных в одном месте разработчиков (около 20) я этой моделью доволен. По мере увеличения числа разработчиков или распределении их по разным местам становится все труднее удерживать всех в одном направлении в отношении таких вещей, как что такое хорошая фиксация или как должны эволюционировать отдельные службы.

Для групп, испытывающих быстрый рост, модель коллективного владения является проблемной. Трудность состоит в том, что для успешной работы коллективного владения вам необходимо время и пространство, которые обеспечивают, чтобы по мере усвоения новых вещей возникал и обновлялся консенсус. Это становится труднее с бо́льшим числом людей в целом и по-настоящему трудно, если вы нанимаете новых людей в быстром темпе (или переводите их в проект).

Потенциальные решения

По моему опыту, сильная степень владения кодом — почти универсальная модель, принятая организациями, имплементирующими крупномасштабные архитектуры на основе микрослужб, состоящие из многочисленных групп и более 100 разработчиков. В каждой группе облегчается принятие решений относительно правил о том, что является хорошим изменением. Вы смотрите на каждую группу как на использующую коллективное владение кодом локально. Эта модель также позволяет создавать группы, которые ориентированы на продукт. Если ваша группа владеет несколькими службами и эти службы ориентированы на бизнес-домен, то ваша группа становится более сосредоточенной на одном участке бизнес-домена. Это облегчает поддержку групп, ориентированных на клиента, которые накапливают доменную компетентность, часто при участии внедренных в них владельцев продуктов, направляющих их работу.

---

[2] Дуршлаг — это миска с большим количеством отверстий, используемая, например, для процеживания макарон.

Переломные изменения

Микрослужба существует как часть более широкой системы. Она либо использует функциональность, предоставляемую другими микрослужбами, либо предоставляет свою собственную функциональность другим потребителям-микрослужбам, либо, возможно, делает и то и другое. С архитектурой на основе микрослужб мы стремимся к независимой развертываемости, но для этого нам нужно обеспечить, чтобы при внесении изменений в микрослужбе мы не ударяли по нашим потребителям.

Мы думаем о функциональности, которую выставляем наружу другим микрослужбам, в терминах *контракта*. Речь идет не только о том, чтобы сказать, что "вот эти данные я буду возвращать". Это также касается определения ожидаемого поведения вашей службы. Заключили ли вы контракт с вашими потребителями явным образом, или нет — совершенно неважно; он просто существует. Когда вы вносите изменения в своей службе, вам нужно обеспечить, чтобы вы не нарушали этот контракт, в противном случае возникнут неприятные производственные проблемы.

Рано или поздно вам придется устранять трудности, причинами которых являются переломные изменения — либо потому, что вы приняли сознательное решение внести обратно несовместимое изменение, либо, возможно, потому, что вы внесли "невинное" изменение, предполагая, что оно повлияет только на вашу локальную службу, в результате обнаружив, что это нарушило другие службы путями, о которых вы даже не представляли.

Как эта проблема проявляется?

Худшее проявление этой трудности возникает, когда вы видите перебои в производстве, вызванные отправкой в "прямой эфир" новых микрослужб, которые нарушают совместимость с существующими службами. Это признак того, что вы не отлавливаете нечаянные нарушения контракта заблаговременно. Подобные трудности бывают катастрофическими, если у вас не установлен механизм быстрого отката. Единственный положительный момент, который можно извлечь из этих режимов сбоя, состоит в том, что они обычно проявляются довольно быстро сразу после релиза, если только обратно несовместимое изменение не было внесено в часть контракта с редко используемой службой.

Еще один признак — вы начинаете видеть, как люди пытаются вместе оркестрировать одновременные развертывания многочисленных служб (иногда это именуется "командно-строевым" релизом). Это также свидетельствует о попытках управлять изменениями контракта между клиентом и службой. Нерегулярный командно-строевой релиз не так уж плох внутри группы, но если он встречается часто, то что-то нужно расследовать.

Когда эта проблема возникает?

Я рассматриваю эту проблему как болезнь роста, с которой группы сталкиваются на довольно ранней стадии, в особенности когда развитие "размазано" по двум и более группам. Внутри одной группы люди более осведомлены, когда вносят пере-

ломные изменения, отчасти потому, что есть хороший шанс, что разработчики трудятся как над изменяемой, так и над потребляющей службами. Когда вы попадаете в ситуации, где одна группа изменяет службу, которая затем потребляется другими группами, эта проблема может возникать чаще.

Со временем, по мере того как группы становятся более зрелыми, они действуют аккуратнее при внесении изменений прежде всего во избежание нарушений, а также создают механизмы для заблаговременного отлавливания проблем.

Потенциальные решения

Для управления случаями нарушения контрактов у меня есть набор правил. И они довольно простые:

1. Не нарушать контракты.
2. См. правило 1.

Ладно, шучу, но только слегка. Вносить переломные изменения в контракты, которые вы выставляете наружу, — не очень замечательная идея, которая к тому же служит источником головной боли при их управлении. На самом деле вы хотите свести их к минимуму, если сможете. Тем не менее вот более реалистичные правила:

1. Устранять "нечаянные" переломные изменения.
2. Хорошенько подумать, прежде чем вносить переломное изменение, — нельзя ли его избежать?
3. Если нужно внести переломные изменения, то следует давать своим потребителям время на миграцию.

Давайте рассмотрим эти шаги подробнее.

Устранять "нечаянные" переломные изменения

Наличие явной схемы для вашей микрослужбы быстро обнаружит структурные нарушения в вашем контракте. Если вы предоставляете метод расчета, который ранее брал в качестве параметров два целых числа, но теперь берет только одно целое число, то очевидно, что такое изменение переломное, и оно должно быть очевидным из новой схемы. Выражение этой схемы совершенно явным образом для разработчиков поможет с ранним обнаружением подобного рода вещей. Если они должны пойти и внести изменения в схему вручную, то это становится явным шагом, который, как мы надеемся, заставит их приостановиться на мгновение и подумать об изменении еще раз. Если у вас есть формат формальной схемы, то, разумеется, также существует вариант уладить этот вопрос программно, хотя это делается не так часто, как хотелось бы. Средство сопровождения протокольных буферов protolock[3] — пример одного из таких инструментов, который фактически запрещает вносить несовместимые изменения в ваши протокольные буферы.

[3] См. **http://bit.ly/2kUxvbq**.

Вариант, принятый многими людьми по умолчанию, состоит в использовании бессхемных форматов обмена, причем наиболее распространенным примером является JSON. Хотя для JSON теоретически можно определять явные схемы, они не применяются на практике. Изначально разработчики склонны проклинать ограничения формальных схем — вынужденные покорпеть над критическими изменениями в разных службах, они поменяют свое мнение. Также следует отметить, что некоторые форматы сериализации с использованием схем улучшают производительность десериализации данных благодаря формальным типам, об этом тоже стоит подумать.

Однако структурные нарушения — лишь часть трудности. Вам также следует учесть семантические нарушения. Если наш метод расчета по-прежнему берет два целых числа, но последняя версия нашей микрослужбы эти два целых числа умножает, тогда как раньше она просто их складывала, то такое изменение также является нарушением контракта. Один из лучших практических способов обнаружить подобное нарушение — тестирование. Мы вскоре его рассмотрим.

Что бы вы ни делали, самый быстрый выигрыш — сделать результат как можно более очевидным для разработчиков, когда они вносят изменения во внешний контракт. Это может означать избегание технологии, которая "волшебным" образом сериализует данные или генерирует схемы из кода, вместо нее предпочитая раскручивать такие вещи вручную. Поверьте — сделать так, чтобы было трудно изменить контракт со службой, лучше, чем постоянно ударять по потребителям.

Хорошенько подумать, прежде чем вносить переломные изменения

Если это возможно, отдавайте предпочтение расширяющим изменениям в вашем контракте со службой. Добавьте новые методы, ресурсы, темы или что-то еще, что поддерживает новую функциональность, не удаляя старую. Старайтесь найти способы поддерживать старое и в то же время поддерживать новое. Это будет означать, что вам придется поддерживать старый код, но эта работа все равно будет меньше, чем улаживание переломного изменения. Помните, что если вы решите нарушить контракт, то именно вы и должны уладить последствия данного решения.

Давать потребителям время на миграцию

Как мне было ясно с самого начала, микрослужбы предназначены для независимого развертывания. При внесении изменений в микрослужбу вам нужно иметь возможность развертывать эту микрослужбу в производстве без необходимости развертывания чего-либо еще. Для того чтобы это работало как надо, вам желательно изменить контракт со службой так, чтобы существующие потребители не были затронуты — отсюда следует, что вам нужно разрешить потребителям по-прежнему использовать старый контракт, даже если у вас имеется новый контракт. Затем вам нужно дать всем потребителям время на то, чтобы изменить свои службы для миграции на новую версию службы.

Я видел два способа, как это делается. Первый — выполнять две версии микрослужбы, как показано на рис. 5.2. Здесь одновременно доступны две сборки службы "Уведомления", каждая из которых выставляет наружу разные несовместимые конечные точки, между которыми потребители могут выбирать. Первостепенные трудности с этим подходом заключаются в том, что для выполнения дополнительных служб вам нужно иметь добавочную инфраструктуру, вероятно, придется поддерживать совместимость данных между версиями служб, и потребуется устранение дефектов для всех работающих версий, что неизбежно повлечет за собой ветвление исходного кода. Описанные проблемы несколько смягчаются, если вы даете сосуществовать двум версиям только короткий период времени, что является единственной ситуацией, где я бы рассматривал этот подход.

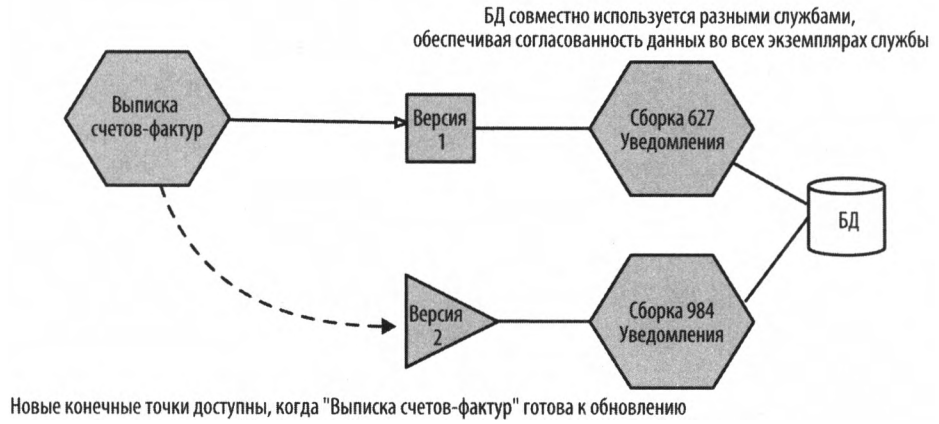


Рис. 5.2. Обеспечение сосуществования двух версий одной и той же микрослужбы для поддержки обратно несовместимых изменений

Способ, который я предпочитаю, состоит в том, чтобы иметь одну работающую версию вашей микрослужбы, но делать так, чтобы она поддерживала оба контракта, как мы видим на рис. 5.3. Возможно, к примеру, предоставление двух API на разных портах. Указанный подход усложняет имплементацию микрослужб, но позволяет избежать трудностей предыдущего способа. Я беседовал с сотрудниками

Рис. 5.3. Одна служба, выставляющая наружу два контракта

некоторых групп, которые поддерживают три или более старых контракта в одной и той же службе годы спустя, из-за того что внешние потребители не способны измениться. Оказаться в такой позиции — не из приятных, но если вы все-таки оказались с потребителями, которые не изменяются, то я по-прежнему считаю, что такой вариант будет самым лучшим.

Конечно, если одна и та же группа улаживает вопросы как с потребителем, так и с производителем, то вы можете выполнить "командно-строевой релиз" и развернуть новые версии как потребителя, так и производителя одновременно. Этот подход не из тех, которые я хотел бы применять часто, но, по крайней мере, внутри одной группы он упрощает управление релизной координацией — просто не превращайте его в привычку!

Изменениями внутри группы легче управлять, т. к. вы контролируете обе стороны уравнения. По мере того как микрослужба, которую вы хотите изменить, используется все шире, стоимость управления изменением становится больше. В результате вы более расслаблены по поводу внесения переломных изменений внутри группы, но нарушение API, который вы выставляете наружу сторонним организациям, скорее всего, будет довольно болезненным.

Как бы вы это ни делали, вам нужна хорошая коммуникация с людьми, которые управляют службами, потребляющими вашу службу. Вы (в лучшем случае), скорее всего, будете создавать им неудобства, поэтому желательно быть с ними в хороших отношениях. Относитесь к потребителям своей службы как к клиентам, а относиться к своим клиентам следует подобающе!

> **Разберись с этим — быстро!**
>
> По мере того как в организациях растет число микрослужб, они в итоге вырабатывают в целом эффективный способ устранения "нечаянных" переломных изменений и придумывают управляемый механизм для работы с целенаправленными изменениями. Если этого не происходит, то последствия становятся настолько значительными, что архитектура на основе микрослужб оказывается несостоятельной. Говоря по-другому, у меня есть сильные подозрения в том, что малые организации, которые базируются на микрослужбах, не разбирающиеся в этом вопросе, не продержатся достаточно долго, чтобы вырасти в крупные организации.

Отчетность

Вместе с монолитной системой в типичной ситуации у вас имеется монолитная база данных. Это означает, что заинтересованные стороны, которые хотят анализировать все данные вместе, что часто предусматривает крупные операции соединения по всем данным, имеют готовую схему, относительно которой они исполняют свои отчеты. Они могут исполнять их непосредственно относительно монолитной базы данных, возможно, относительно реплики для чтения, как показано на рис. 5.4. В архитектуре на основе микрослужб мы "разломали" эту монолитную схему. В результате потребность в отчетности по всем нашим данным не исчезла, мы просто намного затруднили ситуацию, т. к. теперь наши данные разбросаны по нескольким логически изолированным схемам.

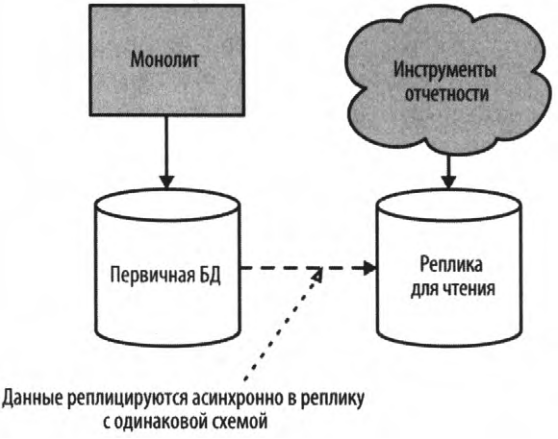


Рис. 5.4. Отчетность проводится непосредственно на базе данных монолита

Когда эта проблема возникает?

Эта проблема стремится "укусить" довольно рано и в обычных условиях приходит на стадии, когда вы начинаете подумывать о декомпозиции монолитной схемы. Надо надеяться, что она будет обнаружена до того, как появятся затруднения, но я видел немало проектов, в которых вплоть до середины проекта группа не понимала, что архитектурное направление будет создавать страдания для сторон, заинтересованных в вариантах с использованием отчетности. Слишком часто потребности в нижестоящей отчетности не рассматриваются заблаговременно, поскольку это происходит вне сферы нормальной разработки софта и сопровождения системы — "с глаз долой, из сердца вон".

Эту проблему можно обойти, если ваш монолит уже использует выделенный источник данных для отчетности, например, хранилище данных или озеро данных. Тогда вам нужно лишь обеспечить, чтобы микрослужбы могли копировать соответствующие данные в существующие источники данных.

Потенциальные решения

Во многих ситуациях заинтересованные стороны, которых волнует вопрос наличия доступа ко всем вашим данным в одном месте, вероятно, также инвестировали в цепочку инструментов и процессы, которые ожидают прямого доступа к базе данных, обычно используя SQL. Из этого также следует, что их отчетность, скорее всего, связана с дизайном схемы вашей монолитной базы данных. Это означает, что если вы не хотите изменить характер их работы, то вам так или иначе нужно будет представить единую базу отчетных данных и, вполне возможно, такую, которая сочетается с дизайном старой схемы, чтобы ограничить влияние любых изменений.

Наиболее прямолинейный подход к решению указанной проблемы — сначала отделить потребность в единой базе данных для хранения данных отчетности от баз данных, которые используются микрослужбами для хранения и извлечения данных,

как показано на рис. 5.5. Это позволяет размежевать содержимое и дизайн базы отчетных данных от дизайна и эволюции требований к хранению данных в каждой службе. Это также дает возможность вносить изменения в указанную новую базу отчетных данных с учетом конкретных требований предоставления отчетности пользователям. От вас нужно лишь выяснить, как ваши микрослужбы будут "вталкивать" данные в новую схему.

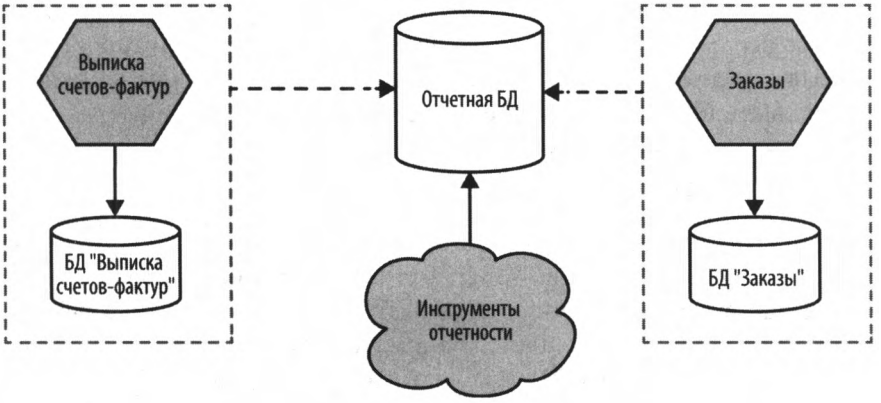


Рис. 5.5. Выделенная база отчетных данных, в которую передаются данные из разных микрослужб

Мы уже рассматривали возможные решения описанной проблемы в *главе 4*. Система захвата изменений в данных является очевидным потенциальным примером решения этой проблемы, но такие методы, как проекции базы данных, также бывают полезными, поскольку вы можете спроецировать единую отчетную схему из проекций, выставленных наружу из схем многочисленных баз данных микрослужб. Вы также можете подумать о других методах, таких, как программное копирование данных в отчетную схему в рамках кода ваших микрослужб, или, возможно, о наличии промежуточных компонентов, которые будут заполнять базу отчетных данных, прослушивая события вышестоящих служб.

Я провожу разведывательный анализ трудностей и потенциальных решений, связанных с этой темой, подробнее в *главе 5* книги "Создание микросервисов".

Мониторинг и устранение неполадок

> "Мы заменили наш монолит микрослужбами, для того чтобы каждый перебой в работе был больше похож на детективную историю убийства".
>
> – *Честная страница статуса* (@honest_update), **http://bit.ly/2mldxqH**

Со стандартным монолитным приложением подход к мониторингу довольно прост. У нас есть небольшое число машин, о которых нужно беспокоиться, а режим сбоя приложения является в некотором роде "двоичным" — приложение часто либо "поднято", либо "лежит". С архитектурой на основе микрослужб мы будем учитывать сбой только одного экземпляра службы или только одного типа экземпляра — можем ли мы их надлежаще восстанавливать?

С монолитной системой, если наш процессор застрял на 100% в течение длительного времени, то мы знаем, что это большая проблема. С архитектурой на основе микрослужб с десятками или сотнями процессов можно ли сказать то же самое? Нужно ли нам будить кого-то в 3 часа ночи, когда только один процесс "застрял" на 100%-ной загруженности CPU?

Выяснение того, где у вас проблемы, и понимание того, действительно ли нужно беспокоиться о той или иной проблеме, которую вы видите, усложняется по мере того, как у вас все больше "движущихся частей". Подход к мониторингу и устранению неисправностей будет меняться по мере роста архитектуры на основе микрослужб. Эта область потребует постоянного внимания и вложений.

Когда эти проблемы возникают?

Предсказать, когда точно у вас возникнут тут проблемы, чуточку сложнее. Простым ответом может быть фраза: "Впервые что-то не так в производстве", но попытки выяснить, что и где пошло не так, будет той работой, которой разработчикам и тестировщикам, возможно, придется заниматься до того, как вы вообще доберетесь до производства. Вы столкнетесь с этими ограничениями, когда у вас будет пара служб, или, пожалуй, не раньше, чем достигнете 20 или более служб.

Поскольку трудно точно предсказать, когда ваш существующий подход к мониторингу начнет вас подводить, можно предложить лишь заблаговременно приоритизировать имплементацию базовых усовершенствований.

Как эти проблемы проявляются?

В некотором смысле их довольно легко заметить. Вы увидите производственные проблемы, которые вы не можете объяснить или понять, у вас будут оповещения, которые будут срабатывать, несмотря на то что система, по всей видимости, является здоровой, и вам будет труднее ответить на простой вопрос "Все ли в порядке?"

Потенциальные решения

Изменить способ мониторинга и устранения неисправностей в архитектуре на основе микрослужб поможет масса механизмов — некоторые из них имплементировать проще, другие же сложнее. Далее приводится неисчерпывающий обзор некоторых ключевых моментов, которые следует учитывать.

Агрегирование журналов

С малым числом машин, в особенности долгоживущих машин, когда нам нужно проверять журналы регистрации операций, мы обычно идем к самим машинам и получаем информацию. Проблема с архитектурой на основе микрослужб состоит в том, что у нас часто работает гораздо больше процессов на большем числе машин, которые бывают короткоживущими (например, виртуальные машины или контейнеры).

Система агрегирования журналов позволит вам захватывать все ваши журналы и пересылать их в центральное место, где в них можно производить поиск, а в некоторых случаях даже генерировать оповещения. Существует много вариантов, от стека ELK с открытым исходным кодом (Elastic search, Logstash/Fluent D и Kibana)[4] до лично мою любимого Humio[5], и эти системы невероятно полезны.

Хорошенько подумайте об имплементации агрегирования журналов в качестве первого шага перед имплементацией архитектуры на основе микрослужб. Это невероятно полезно и является хорошей проверкой способности вашей организации внедрять изменения в операционном пространстве.

Агрегирование журналов — один из самых простых в имплементации механизмов, его следует применять на ранней стадии. На самом деле я считаю, что это первое, что вы должны сделать при имплементации архитектуры на основе микрослужб. Отчасти потому, что это очень полезно с самого начала. Кроме того, если ваша организация испытывает трудности в имплементации подходящей системы агрегирования журналов, то вы, возможно, захотите пересмотреть вопрос о готовности к переходу на микрослужбы. Работа, необходимая для имплементации системы агрегирования журналов, довольно проста, и если вы не готовы к этому как организация, то шаг в сторону микрослужб, вероятно, будет излишним.

Трассировка

Понимание того, где последовательность вызовов между микрослужбами не сработала или какие службы вызвали всплеск задержки, бывает затруднено, если вы анализируете информацию только из каждой изолированной службы. Возможность сопоставлять серию потоков и смотреть на них в целом невероятно полезна.

В качестве отправной точки создайте ИД корреляции для каждого входящего в систему вызова, как показано на рис. 5.6. Когда служба "Счет-фактура" получает вызов, ей присваивается ИД корреляции. Когда она отправляет вызов микрослужбе "Уведомления", она также передает ИД корреляции посредством заголовка HTTP или поля в полезной нагрузке сообщения, или какого-либо другого механизма. Как правило, в части генерирования исходного ИД корреляции я бы посмотрел на API шлюза или сетку для служб.

Когда служба "Уведомления" улаживает вызов, она регистрирует информацию о том, что она делает, в сочетании с тем же ИД корреляции, позволяя использовать систему агрегирования журналов для опроса всех журналов, ассоциированных с данным ИД корреляции (при условии, что вы помещаете ИД корреляции в стандартное место в журнальном формате). Разумеется, с ИД корреляции можно делать и другие вещи, например, управлять сагами (как мы обсуждали в *главе 4*).

Развивая эту идею дальше, мы можем использовать инструменты также для отслеживания времени, затрачиваемого на вызовы. Учитывая, как работают системы агрегирования журналов, где журналы укладываются в пакеты и передаются цен-

[4] См. **https://www.elastic.co/elk-stack**.

[5] См. **https://humio.com/**.

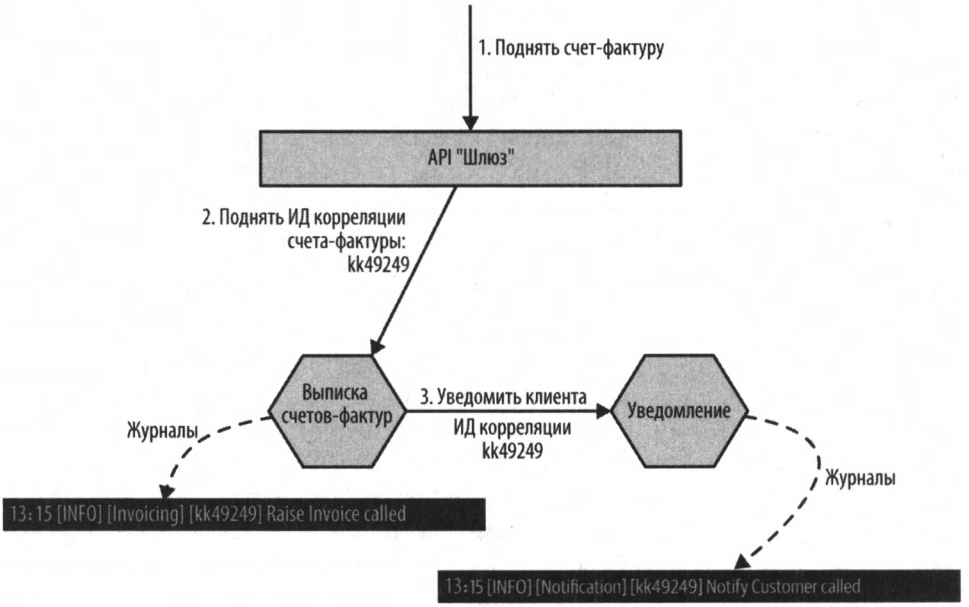


Рис. 5.6. Использование ИД корреляции для обеспечения возможности сбора информации о конкретной цепочке вызовов

тральному агенту на регулярной основе, невозможно получить точную информацию, позволяющую точно определить, где тратится время в течение цепочки вызовов. Здесь помогут системы распределенной трассировки с открытым исходным кодом, подобные Jaeger[6], показанной на рис. 5.7.

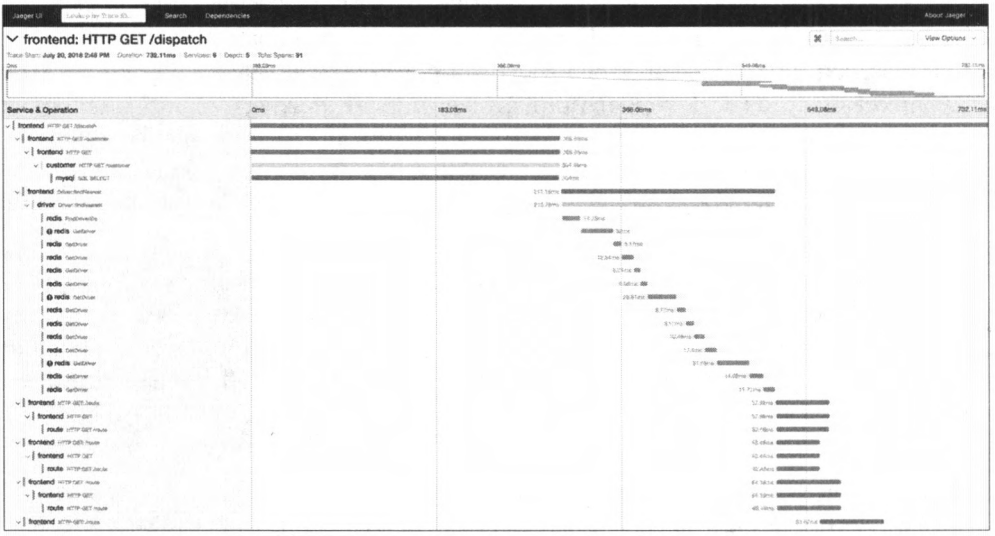


Рис. 5.7. Jaeger — это инструмент с открытым исходным кодом для сбора информации для распределенных трасс и анализа производительности отдельных вызовов

[6] См. **https://www.jaegertracing.io/**.

Чем чувствительнее к задержке ваше приложение, тем скорее я бы стремился имплементировать инструмент распределенной трассировки, такой, как Jaeger. Стоит отметить, что если вы встроили генерирование идентификаторов корреляции и их использование в существующие архитектуры на основе микрослужб (за что я в целом выступаю, при условии, что эта работа делается задолго до того, как вам понадобится инструмент распределенной трассировки), то у вас, по всей видимости, уже есть места в стеке существующих служб, которые можно легко изменить, чтобы заложить данные в подходящий инструмент. Также поможет сетка для служб, поскольку она улаживает для вас, по крайней мере, входящую и исходящую трассировку, даже если она не слишком полезна в части оснащения инструментами обработки вызовов внутри отдельных микрослужб.

Испытание в производстве

Функциональные автоматизированные испытания в типичной ситуации позволяют получить обратную связь перед развертыванием относительно того, обладает ли наш софт достаточными качественными характеристиками для развертывания. Но после того как софт попадает в производство, мы по-прежнему хотим получать ту же обратную связь! Даже если данная функция раньше уже в производстве работала, новое развертывание службы или изменение среды может нарушить эту функциональность позже.

Впрыскивая поведение "поддельного" пользователя в нашу систему в форме того, что часто именуется синтетическими транзакциями, мы можем определять ожидаемое поведение и соответственно оповещать, если оно не является таким. В одной из моих предыдущих компаний, Atomist, у нас был довольно сложный процесс интеграции для новых клиентов, который требовал авторизации нашего софта с помощью их учетных записей GitHub и Slack. Имелось достаточно "движущихся частей", которые на ранней стадии этого процесса сталкивались с трудностями, такими, как ограничение числа запросов к API-интерфейсам GitHub. Один из моих коллег, Сильвен Хеллегуарч (Sylvain Hellegouarch), написал сценарий регистрации "поддельных" клиентов. Для одного из этих "поддельных" клиентов мы на регулярной основе запускали процесс регистрации, который описывал весь процесс от начала до конца. Когда он не срабатывал, это часто было признаком того, что какая-то часть работает не так в наших системах, и ее было гораздо лучше отлавливать с помощью "поддельного" пользователя, а не реального!

Хорошая отправная точка для проведения испытаний в производстве — использование существующих сквозных тестовых случаев и их переработка для применения в производственной среде. Важно обеспечить, чтобы эти "тесты" не оказывали непредвиденного влияния на производство. С помощью Atomist мы создали учетные записи GitHub и Slack, которые контролировали в синтетических транзакциях, поэтому ни один реальный человек не был вовлечен или затронут, и наши сценарии легко очищали указанные учетные записи после этого. С другой стороны, я слышал, как одна компания по ошибке заказала 200 стиральных машин для доставки в свой головной офис, потому что они не учли тот факт, что испытательные заказы действительно будут отправлены. Поэтому будьте осторожны!

В направлении наблюдаемости

С помощью традиционных процессов мониторинга и оповещения мы думаем о том, что может пойти не так, собираем информацию, сообщающую о том, когда это происходит, и используем ее для запуска оповещений. Таким образом, мы в первую очередь настраиваем себя на улаживание известных причин проблем — исчерпание дискового пространства, не отвечающий экземпляр службы или, возможно, всплеск задержки.

По мере того как наши системы усложняются, становится все труднее предсказывать все неприятные пути, которыми наша система может нас подвести. В данной точке важно обеспечить возможность задавать открытые вопросы[7] о наших системах, когда эти трудности возникают, чтобы помочь нам в первую очередь остановить "кровотечение" и убедиться, что система может продолжать работу, а затем позволить нам собирать информацию в достаточном объеме для устранения проблем в будущем.

Таким образом, мы должны быть в состоянии собрать много информации о том, что делает наша система, позволяя нам постфактум задавать вопросы о данных, о которых мы должны были спросить изначально, но о которых мы не знали. Трассировка и журналы служат важным источником данных, на основе которых мы можем формулировать вопросы и использовать реальную информацию, а не догадки с целью определения причины проблемы. Секрет в том, чтобы упростить запрос и просмотр этой информации в контексте.

Не исходите из того, что вы знаете ответы заранее. Скорее, примите точку зрения, что вы будете удивлены, поэтому совершенствуйтесь в постановке вопросов о своей системе и обеспечьте использование цепочки инструментов, которые позволяют выполнять нерегламентированные запросы информации. Если вы хотите разведать эту концепцию подробнее, то я рекомендую книгу "Наблюдаемость распределенных систем" в качестве отличной отправной точки[8].

Локальный опыт разработчиков

По мере того как у вас возникает все больше и больше служб, опыт разработчика начнет страдать. Более ресурсоемкие среды выполнения, такие, как JVM, ограничивают число микрослужб, которые способны выполняться на одном компьютере разработчика. На своем ноутбуке я бы смог, вероятно, запустить четыре-пять микрослужб на базе JVM в качестве отдельных процессов, но смогу ли я запустить десять или двадцать? Скорее всего, нет. Даже с менее обременительными временами выполнения существует ограничение на число вещей, которые вы можете запускать

---

[7] Для справки: открытый вопрос (open-ended question) — это вопрос, который предполагает многовариантный ответ. На него невозможно ответить просто «да» или «нет» — *Пер.*

[8] См. Синди Сридхаран «Наблюдаемость распределенных систем» (Cindy Sridharan, Distributed Systems Observability, Sebastopol: O'Reilly Media, Inc., 2018).

локально, что неизбежно приведет к разговорам о том, что делать, когда вы не можете запустить всю систему на одной машине.

Как эта проблема проявляется?

Процесс разработки изо дня в день начнет замедляться, при этом локальные сборки и исполнение будут занимать больше времени из-за большего числа служб, которые должны быть подняты. Разработчики начнут запрашивать более крупные машины для урегулирования числа служб, которые им приходится улаживать, и, хотя в краткосрочном плане, возможно, все с этим будет в порядке, вам удастся выиграть лишь некоторое время, в случае если имущество служб продолжит расти.

Когда эта проблема возникает?

Момент, когда именно это проявится, скорее всего, зависит от числа служб, которые разработчик хочет выполнять локально, в сочетании с ресурсным следом этих служб. Группа, использующая Go, Node или Python, вполне может обнаружить, что у них локально работает больше служб, прежде чем проявится ограничение ресурсов, а вот группа, использующая JVM, столкнется с этой проблемой раньше.

Я также думаю, что группы, практикующие коллективное владение несколькими службами, более подвержены данной проблеме. Им, скорее всего, потребуется возможность переключаться между разными службами во время их разработки. Группы, имеющие сильную степень владения несколькими службами, будут в основном сосредоточены только на своих собственных службах и, скорее всего, разработают механизмы подавления служб, находящихся вне их контроля.

Потенциальные решения

Если я хочу разрабатывать локально, но уменьшить число служб, которые мне нужно выполнять, то общепринятым методом является установка "заглушек" на те службы, которые я не хочу выполнять сам, или же иметь способ указать их на экземпляры, работающие в другом месте. Чистая настройка удаленной разработки позволяет вам вести разработку с участием многочисленных служб, размещаемых на более вместительной инфраструктуре. Однако она сопровождается трудностями, связанными с необходимостью подключения (что может быть проблемой для удаленных работников или частых путешественников), потенциально более медленными циклами обратной связи с потребностью развертывания софта удаленно, прежде чем вы сможете увидеть его работу, и потенциальным взрывным ростом ресурсов (и связанных с ними затрат), необходимых для среды разработчика.

Telepresence[9] — пример инструмента, который стремится сделать гибридный локальный/удаленный трудовой поток по разработке софта проще для пользователей Kubernetes. Вы можете развивать свою службу локально, но Telepresence способен

[9] См. **https://www.telepresence.io/**.

передавать вызовы других служб в удаленный кластер, давая вам (надо надеяться) лучшее из обоих "миров". Облачные функции Azure также можно выполнять локально, но с подключением к удаленным облачным ресурсам, что позволяет строить службы, состоящие из функций с быстрым локальным трудовым потоком по разработке софта, при этом по-прежнему выполнять их в потенциально обширной облачной среде.

Важно видеть, как меняется опыт разработчика по мере увеличения числа служб — поэтому вам нужны механизмы обратной связи. Вам нужно будет постоянно вкладываться для обеспечения максимальной продуктивности разработчиков по мере увеличения числа служб, с которыми они работают.

Выполнение слишком многого

По мере того как у вас появляется все больше служб и их экземпляров, у вас будет все больше процессов, которые необходимо развернуть, настроить и которыми нужно управлять. Существующие методы управления развертыванием и конфигурацией монолитного приложения плохо масштабируются по мере увеличения числа "движущихся частей", которыми необходимо управлять.

В частности, все большее значение приобретает управление желаемым состоянием. Управление желаемым состоянием — это возможность указывать необходимое число и расположение экземпляров служб, а также обеспечивать их постоянную поддержку. Вы легко можете управлять этим с помощью вашего монолита, задействуя ручные процессы, но процесс не будет хорошо масштабироваться, когда у вас десятки или сотни микрослужб, в особенности если каждой из них требуется другое желаемое состояние.

Как эта проблема проявляется?

Вы начнете видеть увеличивающийся процент времени, затрачиваемого на управление развертываниями и устранение неисправностей, возникающих во время этих развертываний. Ошибки будут совершаться всегда, если процессы опираются на ручные действия, и влияние на распределенные системы "невинных" ошибок иногда трудно предсказать.

По мере добавления дополнительных служб и их экземпляров вам потребуется больше людей для управления мероприятиями, связанными с развертыванием и сопровождением производственного парка. Это приведет к просьбам об увеличении числа людей для поддержки вашей операционной группы или, возможно, к тому, что группа доставки больше времени будет затрачивать на развертывание компонентов.

Когда эти проблемы возникают?

Все дело в масштабе. Чем больше у вас микрослужб и чем больше экземпляров этих микрослужб, тем больше ручных процессов или более традиционных автома-

тизированных инструментов управления конфигурацией, таких, как Chef и Puppet, перестают соответствовать требованиям.

Потенциальные решения

Вам нужен инструмент, который обеспечивает высокую степень автоматизации, позволяет разработчикам в идеале выполнять самообслуживаемое развертывание в облаке и берет на себя автоматическое управление желаемым состоянием.

Для микрослужб предпочтительным инструментом в этом пространстве стал Kubernetes. Он требует от вас контейнеризации ваших служб, но как только вы это сделаете, то сможете посредством Kubernetes управлять развертыванием экземпляров служб на многочисленных машинах, давая возможность масштабироваться для повышения робастности и справляться с нагрузкой (при условии, что у вас достаточно для этого аппаратного обеспечения).

"Ванильный" Kubernetes — совсем не то, что я бы посчитал дружественным к разработчикам. Масса людей работает над более удобными для разработчиков абстракциями, и я надеюсь, что эта работа будет продолжаться. В будущем, я ожидаю, что многие разработчики, которые выполняют софт на Kubernetes, даже не поймут этого, поскольку это станет лишь деталью имплементации. Я склонен видеть, что более крупные организации примут упакованную версию Kubernetes, такую, как OpenShift от RedHat, которая укомплектовывает Kubernetes инструментами, облегчающими работу с корпоративной средой, возможно, беря на себя рычаги управления корпоративной идентичностью и доступом. Некоторые из таких упакованных версий также предоставляют упрощенные абстракции для труда разработчиков.

Если вам посчастливилось находиться в публичном облаке, то вы можете использовать целый ряд вариантов для улаживания развертываний своей архитектуры на основе микрослужб, включая предложения управляемого Kubernetes. Например, AWS и Azure предлагают в этом пространстве несколько вариантов. Я большой поклонник технологии "функции-как-службы" (FaaS), подмножества инструментов, именуемых бессерверными. При наличии подходящей платформы разработчики беспокоятся лишь о коде, а опорная платформа берет на себя бо́льшую часть операционной работы. Хотя нынешний набор предложений FaaS имеет ограничения, они, тем не менее, предлагают перспективу резкого сокращения операционных накладных расходов.

В случае групп, с которыми я работаю и которые уже находятся в публичном облаке, я обычно не начинаю с Kubernetes или подобных контейнерных платформ. Вместо этого я принял подход "сначала без сервера" — попробовать использовать бессерверную технологию, такую, как FaaS, в качестве варианта по умолчанию, ввиду сокращения операционной работы. Если ваша проблема не укладывается в ограничения имеющихся для вас бессерверных продуктов, то поищите другие варианты. Очевидно, что не все проблемные пространства равнозначны, но я чувствую, что если вы уже находитесь в публичном облаке, то вам не всегда понадобится сложная платформа на основе контейнеров, такая, как Kubernetes.

Я реально вижу, как в процессе внедрения микрослужб люди слишком рано тянутся к Kubernetes и тому подобному, часто исходя из того, что это условие предварительное. Следует учесть, что такие платформы, как Kubernetes, превосходно справляются с управлением многочисленными процессами, но вы должны подождать до тех пор, пока у вас не будет достаточное количество процессов, которые начнут перенапрягать ваш текущий подход и технологию. Вы, возможно, обнаружите, что вам требуется всего пять микрослужб и что с этим легко справиться с помощью существующих решений — и в таком случае, все отлично! Не принимайте платформу на основе Kubernetes только потому, что вы видите, что все остальные делают так же. Кстати, то же самое можно сказать и о микрослужбах!

Сквозное тестирование

С любым типом автоматизированного функционального испытания вам придется пойти на тонкий компромисс. Чем больше функций тест исполняет (чем шире его охват), тем больше уверенности вы имеете в своем приложении. С другой стороны, чем больше охват теста, тем больше времени занимает его выполнение и тем труднее будет выяснить, что нарушено в ситуациях, когда он не проходит.

Сквозные тесты любого типа систем находятся на крайнем конце шкалы с точки зрения функциональности, которую они охватывают, и мы привыкли к тому, что их писать и поддерживать проблематичнее, чем меньшие по охвату модульные тесты. Хотя нередко это оправдано, поскольку мы хотим уверенности, которая исходит из того, что сквозной тест использует наши системы таким же образом, как и реальный пользователь.

Но с архитектурой на основе микрослужб "охват" наших сквозных тестов становится очень большим. Теперь нам приходится выполнять тесты в нескольких службах, причем все они должны быть развернуты и надлежаще настроены для тестовых сценариев. Мы также должны быть заранее подготовлены к ложным отрицаниям (false negative), которые возникают, когда проблемы среды, такие, как "умирание" экземпляров служб или сетевые тайм-ауты безуспешных развертываний, приводят к неудаче наших тестов. Я бы сказал, что при выполнении сквозных тестов архитектуры на основе микрослужб мы гораздо более уязвимы к проблемам вне нашего контроля, чем со стандартной монолитной архитектурой.

По мере увеличения охвата тестов вы будете тратить больше времени на борьбу с возникающими проблемами, вплоть до того момента, когда попытки создавать и поддерживать сквозные тесты начинают пожирать огромные промежутки времени.

Как эта проблема проявляется?

Один из признаков проблемы — увеличение набора сквозных тестов и времени на их завершение. Это вызвано тем, что многочисленные группы не уверены в том, какие сценарии охватываются, и добавляют новые "на всякий случай". Вы видите больше сбоев в наборе сквозных тестов, которые не высвечивают затруднения, вызванные вашим кодом, и разработчики часто просто прогоняют тесты снова, чтобы увидеть, что они проходят.

Время, затрачиваемое на сквозные тесты, возрастает все больше и больше, и наступает момент, когда вы начинаете ощущать необходимость увеличить число тестировщиков и, возможно, даже создать отдельную группу по тестированию.

Когда эта проблема возникает?

Эта проблема имеет тенденцию подкрадываться к вам незаметно, но я вижу, что она наиболее остро ощущается в ситуациях, когда отработка разных пользовательских путей следования регулируется многочисленными группами. Чем изолированнее каждая группа в своей работе, тем легче им управлять своими собственными тестами локально. Чем больше вам нужно тестировать межгрупповые потоки, тем проблемнее становятся сквозные, крупные по охвату тесты.

Потенциальные решения

В книге "Создание микросервисов" я описал несколько вариантов, помогающих изменить характер работы с тестированием, и посвятил этому фактически целую главу, но вот краткий итог, который поможет вам стартовать.

Ограничить охват функциональных автоматизированных испытаний

Если вы собираетесь писать тестовые случаи, охватывающие несколько служб, то постарайтесь обеспечить, чтобы такие тесты хранились внутри группы, управляющей этими службами, другими словами, избегайте более крупных по охвату тестов, которые пересекают границы группы. Оставляя владение тестами внутри одной группы, вы облегчаете понимание того, какие сценарии надлежаще охватываются, обеспечиваете разработчикам возможность прогона и отладки тестов и более четко формулируете ответственность за обеспечение прогона и прохождения тестов.

Использовать контракты, обусловливаемые потребителем

Возможно, вы захотите подумать об использовании контрактов, обусловливаемых потребителем (consumer-driven contract, CDC), чтобы исключить потребность в тестовых случаях, требующих пересечения служб. При наличии CDC-контрактов у вас есть потребитель вашей микрослужбы, который определяет свои ожидания относительно того, как ваша служба должна себя вести с точки зрения исполняемой спецификации — теста. Когда вы изменяете свою службу, необходимо обеспечить, чтобы эти тесты по-прежнему проходили.

Поскольку эти тесты определяются с точки зрения потребителя, мы получаем хороший охват для восстановления от нечаянного разрыва контракта. Мы также можем понять требования наших потребителей с их точки зрения и, что очень важно, уяснить, как разные потребители, возможно, хотят от нас разных вещей.

CDC-контракты имплементируются с помощью простого трудового потока по разработке софта, но это делается проще с помощью инструментов, предназначенных для поддержки данного метода. Лучшим примером является, наверное, Pact[10].

Стоит отметить, по моему опыту некоторые группы весьма успешно применяли указанный подход, тогда как другие испытывали трудности в его внедрении. Идея здравая, и я знаю, что она хорошо работает, но я еще не до конца понял трудности, с которыми столкнулись некоторые люди, приняв этот метод на вооружение. Данный метод остается недоиспользуемым в решении действительно трудной проблемы.

Использовать автоматическую ремедиацию релиза и прогрессивную доставку

В автоматизированном тестировании мы обычно пытаемся отыскать проблемы, до того как они повлияют на производство, но бывает, что это оказывается все труднее сделать, поскольку ваша система становится сложнее. Следовательно, стоит потратить усилия на снижение влияния производственных проблем.

Как мы уже говорили в *главе 3*, прогрессивная доставка — это зонтичный термин, обозначающий управление поступательным развертыванием новых версий софта для своих клиентов. Идея заключается в том, что вы оцениваете влияние вашего нового релиза с помощью меньшей группы клиентов, принимая решения о том, когда продолжать откат и продолжить ли вообще или же его отменить. Один из примеров метода прогрессивной доставки — "канареечный" релиз.

Определив приемлемые меры для того, как должна вести себя ваша служба, становится возможным контролировать поступательную доставку в автоматическом режиме. В качестве простого примера можно определить допустимый порог процентиля задержек и частоты встречаемости ошибок, равный 95, и продолжать развертывание только в том случае, если эти меры соблюдены. В противном случае вы автоматически откатываете последнюю версию, получая время для анализа произошедшего.

Многие организации применяют эти методы автоматической ремедиации (исправления) релиза. Netflix, в частности, подробно описала использование данной идеи. Они разработали инструмент Spinnaker по управлению развертыванием, отчасти для того, чтобы помочь контролировать поступательную доставку для своих служб, но есть много других способов воплотить подобные идеи на практике.

Я не говорю, что вам следует рассмотреть возможность заменить тестирование автоматической ремедиацией релиза, просто вы должны думать о том, где получать наилучшую отдачу от своих усилий. В итоге вы вполне можете получить гораздо более робастную систему, вложившись в некоторую работу по выявлению проблем, если они действительно возникают, вместо того чтобы просто сосредоточиваться на устранении возникающих проблем.

---

[10] См. **https://pact.io/**.

Важно отметить, что, хотя эти методы хорошо работают вместе, даже если вы думаете, что пока не готовы к внедрению автоматической ремедиации, по-прежнему есть огромная ценность в имплементации какой-то формы прогрессивной доставки. Даже ручное управление прогрессивной доставкой становится большим шагом вперед от простого развертывания нового софта для всех.

Постоянно уточнять циклы обратной связи относительно качества

Понимание того, как и где вы должны проводить испытания, является постоянной заботой. Вам нужны люди, у которых есть контекст для целостного взгляда на процесс развертывания с целью адаптации того, как и где вы тестируете свое приложение. Это означает наличие людей, которые могут выявлять потребность в добавлении новых тестов для охвата областей системы, где они видят увеличение производственных дефектов, а также могут удалять тесты, когда уже есть охват, в попытке улучшить циклы обратной связи.

Короче говоря, речь идет о балансировке между потребностью в быстрой обратной связи и безопасностью. Вы должны быть готовы выявлять, удалять или заменять неправильный тест, равно как и добавлять новый тест.

Глобальная оптимизация против локальной оптимизации

Если допустить, что вы внедрили модель, в рамках которой группа несет бо́льшую ответственность за принятие локальных решений, возможно, владея всем жизненным циклом микрослужб, которыми они управляют, то вы дойдете до точки, где вам нужно будет сбалансировать локальное принятие решений с более глобальными аспектами.

В качестве примера того, как эта проблема может проявиться, возьмем три группы, которые управляют службами "Выписка счетов-фактур", "Уведомления" и "Исполнение". Группа "Выписка счетов-фактур" решает выбрать Oracle в качестве базы данных, поскольку они хорошо ее знают. Группа "Уведомления" хочет использовать MongoDB, потому что она хорошо подходит для их модели программирования. Между тем в группе "Исполнение" хотели бы использовать PostgreSQL, поскольку она у них уже есть. Когда вы смотрите на каждое решение по очереди, все это имеет смысл, и вы понимаете, как та или иная группа сделала свой выбор.

Однако если вы отступите назад и посмотрите на "крупный план", то вы должны спросить себя, хотите ли вы как организация формировать навыки и платить лицензионные сборы за три базы данных с несколько схожими возможностями. Может быть, вам лучше принять на вооружение только одну базу данных, признав, что она неидеальна для всех, но достаточно хороша для большинства? Без способности видеть, что происходит на локальном уровне, и быть в состоянии помещать это видение в глобальный контекст, как вы вообще сможете принимать такие решения?

Как эта проблема проявляется?

Наиболее распространенный вариант проявления описанной проблемы из всех, которые я встречал, — это ситуация, когда кто-то внезапно понимает, что несколько групп решили одну и ту же проблему по-разному, но совсем не догадывались, что все они пытаются решить один и тот же вопрос. Со временем такой подход становится невероятно неэффективным.

Я помню свой разговор с людьми в REA, компании по недвижимости в Австралии. После многих лет создания микрослужб они дошли до того, что осознали существование массы способов, которыми группы развертывают службы. Это приводило к проблемам, когда люди переходили из одной группы в другую, т. к. им приходилось учиться новому способу ведения дел. Кроме того, стало трудно оправдывать дублирующую работу, которую выполняла каждая группа. В результате они решили потратиться на небольшую работу по формированию общего метода решения этого вопроса.

В типичной ситуации вы узнаете об этих вещах ненароком, возможно, после мимолетного комментария, который вы услышали за обедом. Если у вас есть какая-то межгрупповая техническая группа, например, сообщество практиков, то вы обнаружите подобные проблемы гораздо раньше.

Когда эта проблема возникает?

Эта проблема имеет тенденцию возникать в многогрупповых организациях по прошествии некоторого времени, в особенности в организациях, которые предоставляют группам больше свободы в том, как они выполняют свою работу. Не ожидайте увидеть эту проблему на ранней стадии вашего пути по внедрению микрослужб. Вы, вероятно, начнете с четкого общего понимания того, как все должно делаться. Со временем каждая группа будет все больше сосредоточиваться на своих локальных проблемах и будет оптимизировать способ решения конкретной проблемы, и поэтому стержневой совместный взгляд на ведение дел ("вот так мы ведем дела") начнет смещаться.

Я часто вижу, как эта проблема поднимается и обсуждается после того, как организации прошли период масштабирования. Приток большого числа разработчиков за короткий промежуток времени затрудняет масштабирование обмена нерегламентированной информацией. В результате появляется больше обособленных информационных подразделений, между которыми, возможно, потребуется возвести мосты.

Если вы практикуете коллективное владение службами, то оно, вероятно, поможет избежать или, по крайней мере, ограничить указанные трудности, поскольку коллективное владение службами требует определенной степени согласованности с точки зрения того, как решаются проблемы. Иными словами, если вы хотите коллективного владения, то вам *придется* решить эту проблему, в противном случае, ваше коллективное владение не будет масштабироваться.

Потенциальные решения

Мы уже коснулись некоторых идей, которые помогают в этой области. В *главе 2* мы разведали идею необратимых и обратимых решений, как это снова показано на рис. 5.8. Чем выше стоимость изменений, тем сильнее влияние и тем больше вы будете хотеть более широкого консенсуса в основе принятия решений. Чем меньше влияние, тем легче откатить назад и тем больше решений можно оставить локальным группам.

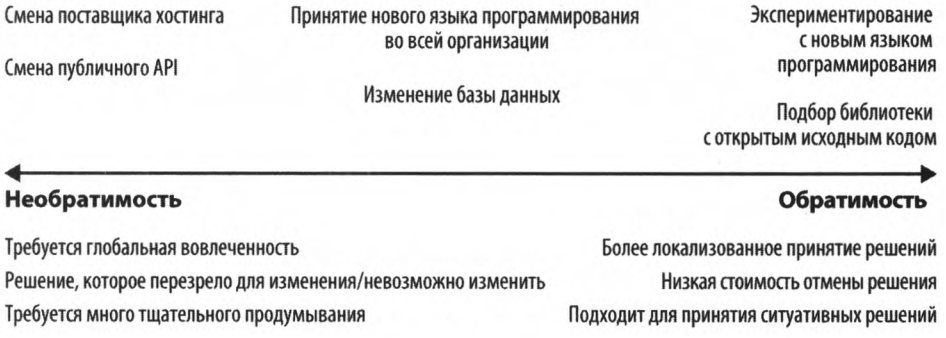


Рис. 5.8. Различия между необратимыми и обратимыми решениями, с примерами вдоль спектра

Хитрость состоит в том, чтобы помочь людям в группе понять, где их решения приведут к необратимым или обратимым концам этого спектра. Чем больше решение тяготеет к необратимости, тем важнее для них будет вовлечение других людей за пределами своей группы в процесс принятия решений. Чтобы это работало как надо, необходимо, по крайней мере, базовое понимание группами более крупноплановых аспектов, для того чтобы видеть, где они могут накладываться, и им также нужна сеть, где они могут выявлять эти трудности и получать отзывы со стороны своих коллег в других группах.

Как простой механизм, разумным подходом является наличие, по крайней мере, одного технического лидера от каждой команды разработчиков, входящего в состав технической сквозной группы, где эти проблемы решаются. Эту группу может возглавлять технический директор, главный архитектор или другое лицо, ответственное за общетехническое видение компании.

Сквозная группа может работать в обоих направлениях. В дополнение к обеспечению места, где группы могут выводить на поверхность локальные проблемы, которые они хотят обсудить на более крупном форуме, это также место, где люди могут поднимать сквозные вопросы. Без некой связи между группами как понять, что на локальном уровне мы решаем проблемы по-разному и что, пожалуй, решение их на глобальном уровне будет иметь больше смысла?

В зависимости от природы вашей организации вы можете рассчитывать на более нерегламентированный, неформальный процесс. Например, в Monzo сотрудники представляют документы в свободной форме, которые внутри организации называются "предложениями". Они публикуются в совместном пространстве, которое,

в свою очередь, оповещает всю компанию через Slack о том, что имеется новое предложение. Заинтересованные стороны могут затем обсудить это предложение и помочь его доработать. Ожидается, что эти предложения не являются законченным артефактом и фактически должны быть открыты для изменений. Это, по-видимому, работает для Monzo как надо, отчасти из-за их культуры относительно коммуникации и распределения ответственности.

В принципе, каждая организация должна отыскать правильный баланс между глобальным и локальным принятием решений. Насколько много ответственности вы будете рады возложить на группы? Сколько контроля вы хотите держать централизованно? Чем больше ответственности вы возлагаете на группы, тем больше вы получаете выгод от большей автономии, но компромисс заключается в том, что у вас будет меньше согласованности в том, как решаются проблемы. Чем больше вы управляете вещами из центра, тем больше вам нужно будет возводить консенсус, и это, вероятно, вас замедлит. Я не могу сказать вам, как достичь баланса между этими двумя силами таким образом, чтобы это было правильно для вас; вам нужно будет самостоятельно выяснить это для себя. Вам просто нужно знать, что такой баланс существует, и вы должны убедиться, что вы собираете правильную информацию для обеспечения возможности настройки этого баланса с течением времени.

Робастность и отказоустойчивость

Распределенные системы демонстрируют целый ряд режимов сбоя, которые могут быть вам незнакомы, если вы больше привыкли к монолитным системам. Сетевые пакеты могут теряться, сетевые вызовы могут истекать, машины могут "умирать" или переставать откликаться. Подобные ситуации бывают редкими в простых распределенных системах, таких, как традиционное монолитное приложение, но по мере увеличения числа служб редкие случаи становятся все более распространенными.

Как эта проблема проявляется?

Подобные проблемы, к сожалению, чаще всего возникают в производственных условиях. Продолжая традиционные циклы разработки и тестирования, мы воссоздаем производственные обстоятельства только на короткие промежутки времени. Эти редкие случаи появляются с меньшей вероятностью, и, когда они все-таки возникают, от них часто отмахиваются.

Когда эта проблема возникает?

Буду здесь честен — если бы я мог сказать вам заранее, когда ваша система будет страдать от нестабильности, то не написал бы эту книгу, т. к., вероятно, проводил бы свое время где-нибудь на пляже, попивая мохито. Могу лишь сказать, что по мере роста числа служб и увеличения числа вызовов служб вы будете становиться все более и более уязвимыми к проблемам отказоустойчивости. Чем более взаимо-

связаны ваши службы, тем вероятнее, что вы будете страдать от таких вещей, как каскадные сбои и встречное (обратное) давление.

Потенциальные решения

Хорошая отправная точка — задать себе пару вопросов о каждом вызове службы, который вы делаете. Во-первых, знаю ли я, каким образом этот вызов может не сработать? Во-вторых, если вызов не сработал, то знаю ли я, что мне делать?

После ответа на эти вопросы вы можете начать искать массу решений. Изолирование служб друг от друга, пожалуй, поможет, включая введение асинхронной коммуникации во избежание временной сопряженности (темы, которую мы затронули в *главе 1*). Использование разумных тайм-аутов позволяет избежать конкуренции за ресурсы с медленными нижестоящими службами, а в сочетании с такими моделями, как прерыватели цепи, вы начнете отказывать быстро[11] во избежание проблем со встречным давлением.

Запуск многочисленных копий служб выручает в случаях с "умиранием" экземпляров, как и платформа, имплементирующая управление желаемым состоянием (которая обеспечивает перезапуск служб при аварийном сбое).

Еще раз повторю мысль, высказанную в *главе 2*, отказоустойчивость — это нечто большее, чем просто имплементация нескольких шаблонов. Речь идет о целом способе выстраивания работы — строительстве организации, которая не только готова справиться с непредвиденными и неизбежно возникающими проблемами, но и по мере необходимости эволюционно формирует рабочие практики. Один из конкретных способов претворить эту идею в жизнь — документировать производственные вопросы, когда они возникают, и вести учет того, что вы усвоили. Очень уж часто я вижу, как организации слишком быстро двигаются дальше, после того как первоначальная проблема была решена или проработана со всех сторон, лишь только для того, чтобы те же самые проблемы вернулись через несколько месяцев.

Честно говоря, я здесь лишь слегка прикоснулся к данной теме. Для более детального знакомства с этими идеями я рекомендую прочитать главу 11 книги "Создание микросервисов", или взглянуть на книгу Майкла Найгарда "Выпускай!" (Release It!, Michael Nygard, Pragmatic Bookshelf, 2018).

"Осиротевшие" службы

Кажется странным, учитывая ряд удивительных технологий, которые у нас есть, и невероятно сложные и масштабируемые системы, которые мы сейчас строим, что мы также по-прежнему встречаем проблемы с некоторыми самыми прозаическими вопросами. На моем опыте, одним из примеров является то, как многие организации с трудом справляются с тем, чтобы точно знать, что у них есть, где оно находится и кто им владеет.

---

[11] Быстрый отказ (fail-fast) в рамках системы раннего обнаружения ошибки означает прерывание работы при возникновении ошибки вместо продолжения работы с ошибкой — *Пер.*

По мере того как микрослужбы будут становиться еще более целенаправленными в своем предназначении, вы обнаружите, что все больше и больше служб успешно работают в течение недель, месяцев или, возможно, лет без каких-либо изменений. С одной стороны, это именно то, что мы хотим. Независимая развертываемость столь привлекательна потому, что позволяет остальной части системы оставаться стабильной, и поддержание стабильности частей нашей системы, которые не нуждаются в изменении, является неплохой идеей.

Я называю эти службы "осиротевшими", поскольку в принципе никто в компании не берет на себя за них ответственность.

Как эта проблема проявляется?

Помню, мне рассказывали истории (возможно, апокрифические) о древних серверах, обнаруженных в старых офисах. Никто не помнил, что они были там, но они по-прежнему счастливо "бегали", делая то, что они делают. Поскольку никто точно не помнил, что делали эти только что обнаруженные компьютеры, люди боялись их выключать. Микрослужбы проявляют некоторые из похожих характеристик. Они существуют, и они (как мы допускаем) работают, но у нас есть та же самая проблема в том, что мы не знаем, что с ними делать, и этот страх может помешать нам их изменить.

Фундаментальная проблема заключается в том, что если некая служба перестает работать или требует изменений, то люди не знают, что делать. Я беседовал с немалым числом групп, члены которых делились историями о том, что они не знали, где был исходный код затрагиваемой службы, и это серьезная проблема.

Когда эта проблема возникает?

Эта проблема обычно возникает в организациях, которые используют микрослужбы в течение длительного периода времени, — достаточно долго, чтобы коллективная память о том, что делает та или иная служба, давно ослабла. Люди, связанные с этой микрослужбой, либо забыли, что они с ней делали, либо, возможно, покинули компанию.

Потенциальные решения

У меня есть (непроверенная) гипотеза о том, что организации, практикующие коллективное владение службами, менее подвержены этой проблеме, прежде всего потому, что им уже пришлось имплементировать механизмы, позволяющие разработчикам переходить от службы к службе и вносить изменения. Такие организации уже, вероятно, ограничивают выбор языка и технологий с целью снижения стоимости переключения контекста между службами. Они также имеют общий инструментарий для внесения изменений в службу, ее тестирования и развертывания. Если общепринятые практики поменялись с момента последнего изменения службы, то, разумеется, это не поможет.

Я говорил с сотрудниками ряда компаний, у которых были трудности подобного рода, и это закончилось созданием простых внутриорганизационных реестров, по-

могающих сопоставлять метаданные служб. Некоторые из реестров просто сканируют репозитории исходного кода, ища файлы метаданных для построения списка имеющихся служб. Эта информация объединяется с реальными данными, поступающими из систем обнаружения служб, таких, как consul или etcd с целью создания более подробной картины того, что работает и с кем вы могли бы об этом поговорить.

В помощь решению этой проблемы Financial Times создала инструмент Biz Ops. Указанная компания имеет несколько сотен служб, разработанных группами по всему миру. Инструмент Biz Ops (рис. 5.9) предоставляет им единое место, где можно найти много полезной информации об их микрослужбах, в дополнение к информации о других ИТ-инфраструктурных службах, таких, как сети и файловые серверы. Построенный поверх графовой базы данных, Biz Ops обладает большой гибкостью в отношении того, какие данные он собирает и каким образом информация модифицируется.

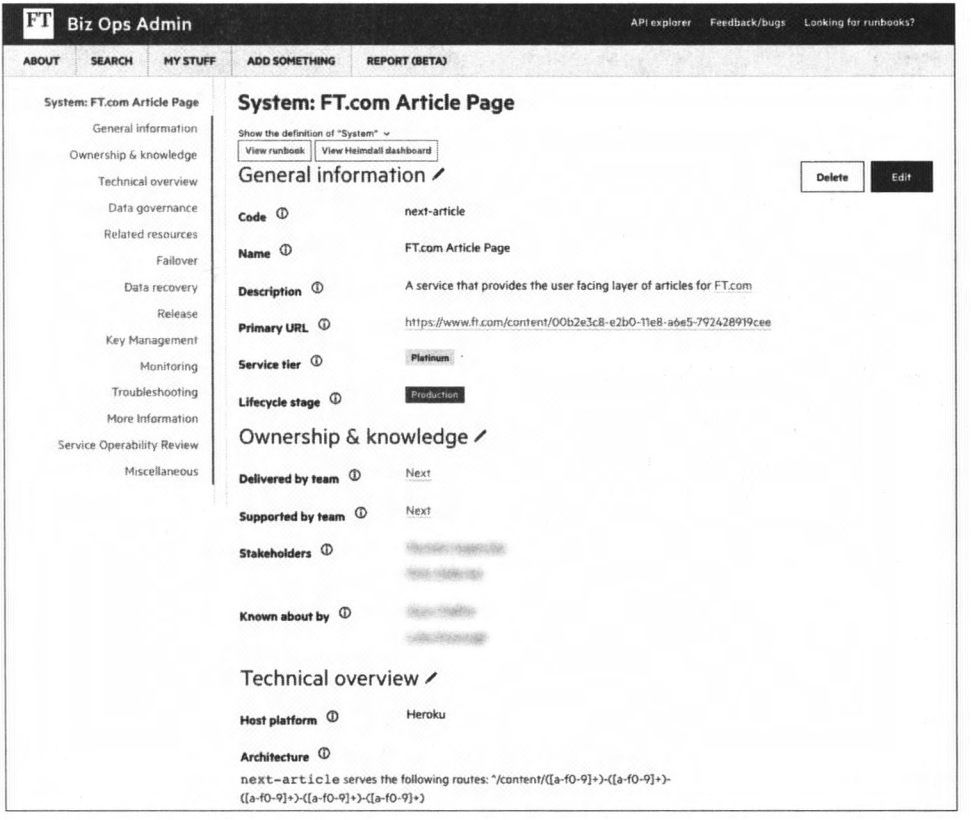


Рис. 5.9. Инструмент Biz Ops компании Financial Times, который собирает информацию о своих микрослужбах

Однако инструмент Biz Ops идет дальше, чем большинство других, которые я видел. Он вычисляет так называемый "балл операбельности" системы, как показано на рис. 5.10. Идея заключается в том, что есть определенные вещи, которые службы и их группы должны делать для обеспечения простоты оперирования службой.

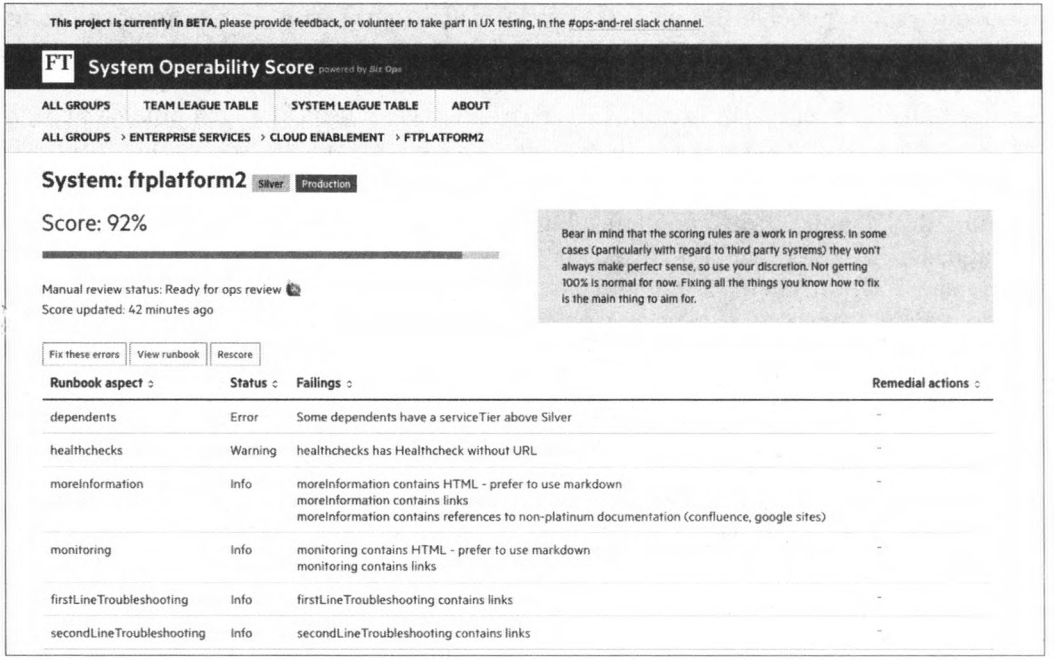


Рис. 5.10. Пример балла операбельности службы для микрослужбы в Financial Times

Сюда входит все, начиная с обеспечения от групп правильной информации в реестре до надлежащей проверки готовности служб к работе. Эти баллы рассчитываются, позволяя группам сразу увидеть, что именно необходимо исправить.

Наличие чего-то вроде реестра служб помогает, но что произойдет, если ваша "осиротевшая" служба предшествует вашему реестру? Главное — выровнять эти недавно обнаруженные "осиротевшие" службы по тому же принципу, как управляются другие службы, а это потребует от вас либо назначить владение существующей группе (если практикуется сильная степень владения), либо развернуть отдельные работы для улучшаемой службы (если практикуется коллективное владение).

Резюме

Материал, приведенный в этой главе, — вовсе не исчерпывающий список всех трудностей, причиной которых являются микрослужбы, и я не перечислил все потенциальные их решения. Вместо этого все внимание было уделено наиболее распространенным проблемам, с которыми, на моем опыте, люди справляются с трудом.

Надеюсь, что я также дал ясно понять, что нет четкого и быстрого правила относительно того, когда именно вы увидите эти проблемы. Каждая ситуация отличается, и в игру вступает масса факторов. Я пытался лишь подчеркнуть, что, хотя вы не способны видеть будущее, вас можно, по крайней мере, предупредить. Сложнее найти правильный баланс между устранением проблем до их возникновения и тратой времени на устранение проблем, которых у вас никогда не будет. Я надеюсь, что прочитав эту главу, вы поймете, какое предупреждение следует ожидать.

Заключение

Итак, мы подошли к концу книги. На всем ее протяжении я надеялся донести до читателя два ключевых сообщения. Во-первых, обеспечьте себе достаточное поле деятельности и соберите правильную информацию для принятия рациональных решений. Не копируйте других, вместо этого подумайте о своей проблеме и своем контексте, оцените варианты и двигайтесь вперед, оставаясь открытыми для изменений, если они вам понадобятся позже. Во-вторых, помните, что поступательное внедрение микрослужб и многих связанных с ними технологий и практик является ключевым фактором. Нет двух одинаковых архитектур на основе микрослужб, хотя есть уроки, которые можно усвоить из работы, проделанной другими, вам нужно найти подход, который хорошо работает в вашем контексте. Разбивая путь на управляемые шаги, вы даете себе наилучшие шансы на успех, поскольку можете адаптировать свой подход по мере продвижения.

Микрослужбы, безусловно, пригодны не для всех. Но, надеюсь, после прочтения этой книги у вас будет не только более точное представление о том, подходят они вам или нет, но и некоторые идеи о том, как начать свое "путешествие".

Библиография

Bird, Christian, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. "Don't Touch My Code! Examining the Effects of Ownership on Software Quality." **http://bit.ly/2p5RlT1** (Не трогай мой код! Обследование влияния владения на качество программно-информационного обеспечения).

Bland, Mike. "Test Mercenaries." **http://bit.ly/2omkxVy** (Легионеры тестирования).

Bland, Mike. "Testing On The Toilet." **http://bit.ly/2ojpWwm** (Тестирование на унитазе).

Brandolini, Alberto. Introducing EventStorming. Leanpub, 2019. **http://bit.ly/2n0zCLU** (Введение в событийный штурм).

Brooks, Frederick P. The Mythical Man-Month, 20th Anniversary Edition. Addison Wesley, 1995 (Мифический человеко-месяц).

Bryant, Daniel. "Building Resilience in Netflix Production Data Migrations: Sangeeta Handa at QCon SF." **http://bit.ly/2m1EwHT** (Укрепление отказоустойчивости во время миграций производственных данных Netflix).

Devops Research & Assessment. Accelerate: State Of Devops Report 2018. **http://bit.ly/2nPDNLe** (Исследование и оценивание Devops. Ускорение. Отчет о состоянии дел в области Devops за 2018 год).

Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003 (Доменно-обусловленный дизайн: пути решения сложности в центре программно-информационного обеспечения).

Feathers, Michael. Working Effectively with Legacy Code. Prentice-Hall, 2004 (Эффективная работа с унаследованным кодом).

Fowler, Martin. "Strangler Fig Application." **http://bit.ly/2p5xMKo** (Приложение "фикус-удавка").

Fowler, Martin. "Reporting Database." **http://bit.ly/2kWW9Ir** (База отчетных данных).

Garcia-Molina, Hector, and Kenneth Salem. "Sagas." ACM Sigmod Record 16, No. 3 (1987): pp. 249–259 (Саги).

Garcia-Molina, Hector, Dieter Gawlick, Johannes Klein, Karl Kleissner, Kenneth Salem. "Modeling Long-Running Activities as Nested Sagas." Data Engineering 14, No. 1 (March 1991): p.14–18 (Моделирование длительных операций как вложенных саг).

Helland, Pat. "Life Beyond Distributed Transactions." Acmqueue 14, No. 5 (Жизнь за пределами распределенных транзакций).

Hodgson, Peter. "Feature Toggles (aka Feature Flags)." **http://bit.ly/2m316zB** (Реле функций (ака флажки функций).

Hohpe, Gregor, and Bobby Woolf. Enterprise Integration Patterns. Addison-Wesley, 2003 (Шаблоны интеграции предприятия).

Humble, Jez, and David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley, 2010 (Непрерывная доставка: надежные релизы программно-информационного обеспечения посредством сборки, тестирования и автоматизации развертывания).

Humble, Jez. "Make Large-Scale Changes Incrementally with Branch by Abstraction." **http://bit.ly/2p95lv7** (Вносите крупномасштабные изменения с помощью ветвления по абстракции).

Kim, Gene, Patrick Debois, Jez Humble, and John Willis. The Devops Handbook. IT Revolution Press, 2016 (Справочник по Devops).

Kleppmann, Martin. Designing Data-Intensive Applications. O'Reilly, 2017 (Дизайн приложений с интенсивным использованием данных).

Kniberg, Henrik, and Anders Ivarsson. "Scaling Agile @ Spotify." October 2012. **http://bit.ly/2ogAz3d** (Масштабирование Agile @ Spotify).

Kotter, John P. Leading Change. Harvard Business Review Press, 1996 (Во главе перемен).

Mitchell, Lorna Jane. PHP Web Services, Second Edition. O'Reilly, 2016 (Веб-службы PHP, 2-е издание).

Newman, Sam. Building Microservices. O'Reilly, 2015 (Создание микросервисов).

Nygard, Michael T. Release It!: Design and Deploy Production-Ready Software, Second Edition. Pragmatic Bookshelf, 2018 (Выпускай!).

Parnas, David. "On the Criteria to be Used in Decomposing Systems into Modules." Information Distributions Aspects of Design Methodology, Proceedings of IFIP Congress '71 (1972) (О критериях, используемых при декомпозиции систем на модули).

Parnas, David. "The Secret History of Information Hiding." David Parnas. In Software Pioneers, edited by M. Broy and E. Denert. (Berlin: Springer, 2002) (Тайная история сокрытия информации).

Pettersen, Snow. "The Road to an Envoy Service Mesh." **https://squ.re/2nts1Gc** (Дорога к сетке для служб в Envoy).

Skelton, Matthew, and Manuel Pais. Team Topologies. IT Revolution Press, 2019 (Топологии групп).

Smith, Steve. "Application Pattern: Verify Branch By Abstraction." **http://bit.ly/2mLVevz** (Шаблон приложения: верифицировать ветвление по абстракции).

Sridharan, Cindy. Distributed Systems Observability. O'Reilly, 2018. **http://bit.ly/2nPZ73d** (Наблюдаемость распределенных систем).

Thorup, Kresten. "Riak on Drugs (and the Other Way Around)." **http://bit.ly/2m1CvLP** (Riak на таблетках (и наоборот)).

Vernon, Vaughn. Domain-Driven Design Distilled. Addison-Wesley, 2016 (Основы доменно-обусловленного дизайна).

Woods, David. "Four concepts for resilience and the implications for the future of resilience engineering." Reliability Engineering & System Safety 141 (2015): 5–9 (Четыре концепции отказоустойчивости и последствия для будущего инженерии отказоустойчивости).

Yourdon, Edward, and Larry Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice-Hall, 1979 (Структурный дизайн: основы дисциплины дизайна компьютерных программ и систем).

Указатель шаблонов

Наименование	Оригинальное название	Описание
Библиотека статических справочных данных	Static reference data library	Перенос статических справочных данных в библиотеку или конфигурационный файл, упаковываемых с каждой микрослужбой, которая в них нуждается
Ветвление по абстракции	Branch by abstraction	Сосуществование двух имплементаций одной и той же функциональности в одной и той же кодовой базе одновременно, позволяя поступательно разрабатывать новую имплементацию, пока она не сможет заменить старую имплементацию
Выделенная схема справочных данных	Dedicated reference data schema	Создание выделенной базы данных для размещения всех статически справочных данных. К указанной базе данных могут обращаться разные службы
Дублирование статических справочных данных	Duplicate static reference data	Копирование статических справочных данных в базы данных микрослужб
Захват изменений в данных	Change data capture	Передача изменений, вносимых в базовое хранилище данных, другим заинтересованным сторонам
Интерфейс "база-данных-как-служба"	Database-as-a-Service interface	Использование выделенной базы данных для предоставления доступа только для чтения к внутренним данным службы
Композиция пользовательского интерфейса	UI composition	Представление единого пользовательского интерфейса путем компоновки большого числа мелких деталей вместе
Монолит как слой доступа к данным	Monolith as data access layer	Доступ к данным, управляемым монолитом через API-интерфейсы, вместо прямого доступа к базе данных
Монолит с выставлением агрегата наружу	Aggregate exposing monolith	Выделение доменных агрегатов наружу из монолита, позволяя микрослужбам обращаться к сущностям, управляемым монолитом
Мультисхемное хранение	Multischema storage	Управление данными в разных базах данных преимущественно при миграции из совместной базы данных в модель "одна база данных — одна служба"

(окончание)

Наименование	Оригинальное название	Описание
Один репозиторий, один ограниченный контекст	Repository per bounded context	Разбиение единого репозиторного слоя вокруг разных частей домена, упрощая декомпозицию на службы
Параллельное выполнение	Parallel run	Выполнение двух имплементаций одной и той же функциональности бок о бок для обеспечения требуемого поведения новой функциональности
Перенос внешнего ключа в код	Move foreign key to code	Перенос управления и обеспечение соблюдения связи по внешнему ключу из одной базы данных на ярус вверх вашей службы
Приложение "фикус-удавка"	Strangler fig application	Обертывание вашей новой архитектуры на основе микрослужб вокруг существующего монолита. Вызовы функциональности, которая была мигрирована из монолита в ваши микрослужбы, перенаправляются; другие вызовы остаются неизменными
Проекция базы данных	Database view	Из опорной базы данных берется проекция, позволяя скрывать части базы данных
Разбиение таблицы	Split table	Разбиение таблицы на две части перед декомпозицией на службы
Синхронизация данных в приложении	Synchronize data in application	Синхронизация данных между двумя источниками истины изнутри одного приложения
Служба "обертывания" базы данных	Database wrapping service	Размещение фасадной службы перед существующей совместной базой данных, позволяя службам мигрировать в сторону от прямого использования базы данных
Служба статических справочных данных	Static reference data service	Выделенная служба, которая предоставляет доступ к статическим справочным данным
Смена владельца данных	Change data ownership	Перенос источника истины из монолита в микрослужбу
Совместная база данных	Shared database	Использование одной базы данных совместно более чем одной службой
Сотрудник-декоратор	Decorating collaborator	Активирование функциональности, работающей в отдельной микрослужбе, путем "вынюхивания" запросов, отправляемых из монолита, и откликов, отправляемых назад в ответ
Трассировочная запись	Tracer write	Поступательная миграция данных из одного источника истины в другой, допуская два источника истины во время миграции

Предметный указатель

Д

З

И

Й

К

М

Н

О

П

P

C

Т

Ф

Х

Ц

Ш

Э

Я

O'REILLY®

ОТ МОНОЛИТА К МИКРОСЕРВИСАМ

Как распутать монолитную систему и мигрировать на микросервисы? Как это сделать, поддерживая работу организации в обычном режиме? В качестве дополнения к чрезвычайно популярной книге Сэма Ньюмена «Создание микросервисов» его новая книга подробно описывает проверенный метод перевода существующей монолитной системы на архитектуру микросервисов.

Это практическое руководство содержит ряд наглядных примеров и шаблонов миграции, массу практических советов по переводу монолитной системы на платформу для микросервисов, различные сценарии и стратегии успешной миграции, начиная с первичного планирования и заканчивая декомпозицией приложений и баз данных. Описанные шаблоны и методы опробованы и надежны, их можно использовать для миграции уже существующей архитектуры.

Книга:

- Идеально подходит для организаций, которые хотят перейти на микросервисы, не занимаясь перестройкой.
- Помогает компаниям определяться с тем, следует ли мигрировать, когда и с чего начинать.
- Решает вопросы коммуникации, интеграции и миграции унаследованных систем.
- Обсуждает несколько шаблонов миграции и мест их применения.
- Рассматривает примеры миграции баз данных, а также стратегии синхронизации.
- Описывает декомпозицию приложений, включая несколько архитектурных шаблонов рефакторизации.
- Раскрывает детали декомпозиции базы данных, включая влияние нарушения ссылочной и транзакционной целостности, новые режимы сбоя и многое другое.

«В книге «От монолита к микросервисам» Сэм Ньюмен дает четкое видение миграции на микросервисы, показывает, каких ловушек (как очевидных, так и коварных) следует остерегаться, и рассматривает ряд очень полезных шаблонов для организационных, архитектурных и технологических изменений» — Дэниел Брайант, технический консультант Datawire и InfoQ.

Сэм Ньюмен принимал участие в нескольких стартапах, 12 лет проработал в ThoughtWorks и теперь является независимым консультантом. Специализируется на микросервисах и облачной архитектуре, проводит обучение и дает консультации, помогает клиентам быстрее и надежнее разрабатывать программно-информационное обеспечение, выступает на конференциях по всему миру. Автор известной книги «Создание микросервисов» издательства O'Reilly.

ISBN 978-5-9775-6723-7

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

bhv®