Оглавление

Предисловие

Разработка хороших, конкурентных и распределенных приложений – сложная задача. Завершив очередной проект на Java, в котором потребовалось использовать массу кода для управления низкоуровневыми потоками выполнения, я задался целью найти более простой инструмент для реализации следующего проекта, который обещал быть еще более сложным.

В марте 2010 г. я увидел твит Дина Вамплера (Dean Wampler), который заставил меня обратить внимание на Akka:

```
W00t! RT @jboner: #akka 0.7 is released: http://bit.ly/9yRGSB
```

После недолгого изучения исходного кода и создания прототипа мы решили использовать Akka. Мы сразу заметили, что новая модель программирования действительно упрощает решение задач, с которыми мы испытали немало сложностей в предыдущем проекте.

Я убедил Роба Баккера (Rob Bakker) совершить со мной увлекательное путешествие в мир новой ультрасовременной технологии, и мы вместе отважно принялись за реализацию первого проекта с применением Scala и Akka. Мы сразу же обратились к Джонасу Бонеру (Jonas Bonér, создатель Akka) за помощью и, как выяснилось потом, оказались первыми широко известными пользователями Akka. Мы завершили этот проект, за которым последовало множество других, и всякий раз убеждались в преимуществах использования Akka.

В ту пору в Интернете было мало информации об Akka, поэтому я решил начать вести блог об этом фреймворке и таким способом внести свой вклад в его развитие.

Я был очень удивлен, когда мне предложили написать эту книгу. Я спросил у Роба, хочет ли он присоединиться ко мне. Позже мы поняли, что нам не обойтись без посторонней помощи, и пригласили Роба Уильямса (Rob Williams). К тому моменту он уже имел опыт создания проектов на Java и Akka.

Мы рады, что наконец смогли закончить эту книгу, описывающую версию Akka (2.4.9), которая включает исчерпывающий набор инструментов для создания распределенных и конкурентных приложений. Мы благодарны читателям, участвующим в программе MEAP (Manning Early Access Program) издательства Manning, за их отзывы. Также для нас, начинающих авторов, бесценной оказалась поддержка издательства Manning Publications.

Всех нас объединило понимание, полученное с опытом работы до использования Akka, что для разработки распределенных и конкурентных приложений на JVM необходим простой и надежный инструмент. Надеемся, нам удастся убедить вас, что Akka является именно таким инструментом.

Раймонд Рестенбург

Благодарности

Нам потребовалось много времени, чтобы написать эту книгу. На всем его протяжении нам помогало множество людей, и мы благодарны им за эту помощь. Я благодарю всех читателей, участвовавших в программе MEAP и купивших ранний выпуск этой книги, за их отзывы, которые помогли значительно улучшить книгу, и за их терпение в течение нескольких лет. Мы надеемся, что вам понравится конечный результат и вы многому научились, участвуя в программе MEAP.

Отдельное спасибо членам проекта Akka, в особенности Джонасу Бонеру (Jonas Bonér), Виктору Клангу (Viktor Klang), Роланду Куну (Roland Kuhn), Патрику Нордвалу (Patrik Nordwall), Бъерну Антонссону (Björn Antonsson), Эндре Варга (Endre Varga) и Конраду Малавски (Konrad Malawski) – все они вдохновляли нас и внесли свой бесценный вклад в книгу.

Мы также хотим поблагодарить Эдвина Рестенбурга (Edwin Roestenburg) и компанию CSC Traffic Management из Нидерландов, доверивших нам начать использовать Akka в важнейших проектах и давших невероятную возможность получить первый опыт с Akka. Мы также хотим сказать спасибо компании Xebia, позволившей Рею в рабочие часы писать книгу и предоставившей потрясающее рабочее место для экспериментов с Akka.

Мы благодарим издательство Manning Publications за оказанное нам доверие. Это наша первая книга, поэтому мы знаем, что это было рискованное предприятие для них. Мы хотим поблагодарить следующих сотрудников Manning за их превосходный труд: Майка Стефенса (Mike Stephens), Джеффа Блейла (Jeff Bleiel), Бена Берга (Ben Berg), Энди Кэрролл (Andy Carroll), Кевина Салливана (Kevin Sullivan), Кэти Теннант (Katie Tennant) и Дотти Марсико (Dottie Marsico).

Мы благодарим Дуга Уоррена (Doug Warren), выполнившего техническую корректуру всех глав. А также многих рецензентов, давших нам ценные советы во время работы над книгой: Энди Хикса (Andy Hicks), Дэвида Гриффита (David Griffith), Дюшана Кайсела (Dušan Kysel), Иэна Старкса (Iain Starks), Джереми Пьерре (Jeremy Pierre), Кевина Эслера (Kevin Esler), Марка Янссена (Mark Janssen), Майкла Шлейхардта (Michael Schleichardt), Ричарда Джеппса (Richard Jepps), Робина Перси (Robin Percy), Рона Ди Франго (Ron Di Frango) и Уильяма Е. Вилера (William E. Wheeler).

Напоследок, но не в последнюю очередь, мы хотим сказать спасибо всем, кто значит для нас больше всего на свете и поддерживал нас во время работы над книгой. Рей благодарит свою жену Шанель (Chanelle), а Роб Уильямс – свою маму, Гейл (Gail) и Лауру (Laurie).

О книге

В этой книге рассказывается о фреймворке Akka и описываются его наиболее важные модули. Мы сосредоточимся на модели программирования с акторами и на модулях поддержки акторов, часто используемых при создании конкурентных и распределенных приложений. На протяжении книги мы постоянно будем показывать, как тестировать код, что является важным аспектом повседневного труда разработчика программного обеспечения. Во всех наших примерах мы будем использовать язык программирования Scala.

После знакомства с основами программирования и тестирования акторов мы рассмотрим все важные аспекты, с которыми вам придется столкнуться при использовании фреймворка Akka в приложениях.

Кому адресована эта книга

Эта книга адресована всем, кто желает узнать, как создавать приложения с Akka. Примеры написаны на Scala, поэтому мы предполагаем, что вы уже имеете некоторое знакомство с этим языком программирования или желаете освоить его в процессе чтения. Также предполагается, что вы знакомы с языком Java, особенно если учесть, что Scala действует поверх JVM.

Содержание книги

Книга состоит из семнадцати глав.

Глава 1 знакомит с акторами Akka. Здесь вы узнаете, как модель программирования с акторами решает некоторые ключевые проблемы, которые традиционно осложняют масштабирование приложений.

Глава 2 сразу же погружается в пример HTTP-службы, реализованной с применением Akka, чтобы показать, как быстро можно создать действующую службу и запустить ее в облаке. Она позволит вам понять, о чем рассказывается в последующих главах.

Глава 3 рассказывает о модульном тестировании акторов с использованием ScalaTest и модуля akka-testkit.

Глава 4 объясняет, как мониторинг акторов позволяет создавать надежные, отказоустойчивые системы.

Глава 5 знакомит с объектами `Future`, чрезвычайно удобным и простым инструментом объединения результатов функций, выполняющихся асинхронно. Здесь вы также узнаете, как объединить акторы и объекты `Future`.

Глава 6 рассказывает о модуле akka-remote, позволяющем взаимодействовать с распределенными акторами по сети. Здесь вы также узнаете, как осуществлять модульное тестирование распределенной системы акторов.

Глава 7 объясняет, как для настройки Akka использовать библиотеку Typesafe Config Library. В ней также рассказывается, как можно использовать эту библиотеку для настройки компонентов вашего приложения.

Глава 8 описывает шаблоны структуризации приложений на основе акторов. Здесь вы узнаете, как реализовать пару классических шаблонов интеграции корпоративных приложений.

Глава 9 объясняет, как применять маршрутизаторы. Маршрутизаторы можно использовать для переключения, широковещательной рассылки и балансировки сообщений между акторами.

Глава 10 знакомит с каналами сообщений, которые можно использовать для передачи сообщений от одного актора другому. Здесь вы познакомитесь с каналами вида точка-точка и публикация/подписка, а также с каналами для недоставленных сообщений и с каналами гарантированной доставки.

Глава 11 обсуждает вопросы конструирования конечных автоматов акторов с использованием модуля FSM и знакомит с агентами, которые можно использовать для асинхронной передачи состояния.

Глава 12 объясняет приемы интеграции с другими системами. В этой главе вы узнаете, как организовать поддержку различных протоколов с помощью Apache Camel и как сконструировать http-службу с применением модуля akka-http.

Глава 13 знакомит с модулем akka-stream. Здесь вы узнаете, как с использованием Akka конструировать потоковые приложения. В этой главе подробно описывается процесс создания потоковой HTTP-службы для обработки событий в журнале.

Глава 14 объясняет, как пользоваться модулем akka-cluster. Здесь вы узнаете, как динамически масштабировать акторы в сетевом кластере.

Глава 15 знакомит с модулем akka-persistence. В этой главе вы узнаете, как записывать и восстанавливать состояние с использованием хранимых акторов, как использовать кластерные расширения для создания приложения покупательской корзины в кластере.

Глава 16 обсуждает ключевые аспекты производительности систем акторов и дает советы по анализу проблем, связанных с производительностью.

Глава 17 заглядывает вперед и знакомит с двумя грядущими нововведениями, которые, как нам кажется, приобретут особую важность: модулем

akka-typed, который сделает возможным проверку сообщений акторов на этапе компиляции, и модулем akka-distributed-data, который обеспечивает распределение состояния в памяти кластера.

Соглашения об оформлении программного кода

Весь исходный код в листингах или в тексте оформлен моноширинным шрифтом, чтобы выделить его на фоне обычного текста. Многие листинги сопровождаются примечаниями и комментариями, подчеркивающими важные понятия. Код примеров из этой книги доступен для загрузки на веб-сайте издательства Manning www.manning.com/books/akka-in-action и в репозитории GitHub https://github.com/RayRoestenburg/akka-in-action.

Требования к программному обеспечению

Все примеры написаны на языке Scala. Они были протестированы с версией Scala 2.11.8. Найти дистрибутив Scala можно здесь: http://www.scala-lang.org/download/.

Обязательно установите последнюю версию sbt (0.13.12 на момент написания этих строк); если у вас установлена более старая версия sbt, вы рискуете столкнуться с проблемами. Загрузить дистрибутив sbt можно отсюда: http://www.scala-sbt.org/download.html.

Фреймворк Akka версии 2.4.9 требует наличия Java 8, поэтому вам также придется установить эту версию Java. Ее можно найти по адресу: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.

Автор в сети

Одновременно с покупкой «Akka в действии» вы получаете бесплатный доступ к частному веб-форуму, организованному издательством Manning Publications, где можно оставлять комментарии о книге, задавать технические вопросы, а также получать помощь от автора и других пользователей. Чтобы получить доступ к форуму и зарегистрироваться на нем, откройте в веб-браузере страницу https://www.manning.com/books/akka-inaction. Здесь описывается, как попасть на форум после регистрации, какие виды помощи доступны и правила поведения на форуме.

Издательство Manning обязуется предоставить своим читателям место встречи, где может состояться содержательный диалог между отдельными читателями и между читателями и автором. Но со стороны автора отсутствуют какие-либо обязательства уделять форуму какое-то определенное внимание – его присутствие на форуме остается добровольным (и неоплачиваемым). Мы предлагаем задавать автору стимулирующие вопросы, чтобы его интерес не угасал!

Форум и архивы предыдущих обсуждений будут доступны на сайте издательства, пока книга находится в печати.

Об авторах

Раймонд Рестенбург (Raymond Roestenburg) – опытный разработчик, программист-полиглот и архитектор ПО. Активный член сообщества Scala, внештатный разработчик Akka, принимавший участие в разработке модуля Akka-Camel.

Роб Баккер (Rob Bakker) – опытный разработчик ПО, занимающийся созданием серверных систем и их интеграцией. Использует Scala и Akka начиная с версии 0.7.

Роб Уильямс (Rob Williams) – основатель онтометрики (ontometric), практики, ориентированной на Java-решения, включая машинное обучение. Впервые использовал акторы десять лет тому назад и на их основе реализовал несколько проектов.

Об иллюстрации на обложке

Иллюстрация с изображением китайского императора на обложке «Akka в действии» взята из книги «Collection of the Dresses of Different Nations, Ancient and Modern» (Коллекция костюмов разных народов, античных и современных) Томаса Джефферис (Thomas Jefferys), опубликованной в Лондоне между 1757 и 1772 годом. На титульной странице указано, что это выполненная вручную каллиграфическая цветная гравюра, обработанная гуммиарабиком.

Томас Джефферис (1719–1771) носил звание «Географ короля Георга III». Английский картограф, был ведущим поставщиком карт того времени. Он выгравировал и напечатал множество карт для нужд правительства, других официальных органов и широкий спектр коммерческих карт и атласов, в частности Северной Америки. Будучи картографом, интересовался местной одеждой народов, населяющих разные земли, и собрал блестящую коллекцию различных платьев в четырех томах.

Очарование далеких земель и дальних путешествий для удовольствия было относительно новым явлением в конце XVIII века, и коллекции, такие как эта, были весьма популярны, знакомя с внешним видом жителей других стран. Разнообразие рисунков, собранных Джефферисом, свидетельствует о проявлении народами мира около 200 лет яркой индивидуальности и уникальности. С тех пор стиль одежды сильно изменился, и исчезло разнообразие, характеризующее различные области и страны. Теперь трудно отличить по одежде даже жителей разных континентов. Если взглянуть на это с оптимистической точки зрения, мы пожертвовали культурным и внешним разнообразием в угоду разнообразию личной

жизни, или в угоду более разнообразной и интересной интеллектуальной и технической деятельности.

В наше время, когда трудно отличить одну техническую книгу от другой, издательство Manning проявило инициативу и деловую сметку, украшая обложки книг изображениями, основанными на богатом разнообразии жизненного уклада народов двухвековой давности, придав новую жизнь рисункам Джефферис.

Глава 1

Введение в Akka

В этой главе:

- почему масштабирование вызывает сложности;
- написано однажды, масштабируется как угодно;
- введение в модель программирования с акторами;
- акторы Akka;
- что такое Akka?

Вплоть до середины 90-х, как раз незадолго до интернет-революции, было общепринято создавать приложения, выполняющиеся на единственном компьютере с единственным процессором. Если приложение оказывалось недостаточно быстродействующим, обычным решением было подождать, пока появится компьютер с более мощным процессором. Проблема решалась сама собой, без изменения исходного кода. Программисты по всему миру таскали бесплатный сыр, и жизнь была прекрасна.

В 2005 г. Херб Саттер (Herb Sutter) написал в журнале *Dr. Dobb's Journal* о необходимости фундаментальных изменений (http://www.gotw.ca/publications/concurrency-ddj.htm). Суть статьи в том, что достигнут физический предел увеличения тактовой частоты процессора и бесплатный обед закончился.

Если приложениям нужно обрабатывать больше данных или обеспечить поддержку большего числа пользователей, они должны стать *конкурентными*. (Строгое определение этого термина мы дадим позже, а пока просто считайте его аналогом термина *многопоточные*. На самом деле это не совсем верно, но на данный момент вполне приемлемая аналогия.)

Масштабируемость – это мера способности системы адаптироваться к изменению спроса на ресурсы без отрицательного влияния на производительность. *Конкуренция* – это средство достижения масштабируемости: предполагается, что при необходимости в серверы могут быть добавлены дополнительные процессоры, и приложение автоматически начнет их

использовать. Это следующее отличное место, откуда можно таскать бесплатный сыр.

Примерно в 2005 г., когда Херб Саттер написал свою замечательную статью, можно было найти компании, запускавшие приложения на кластерах из многопроцессорных серверов (часто их число не превышало двух-трех, на случай, если один из них выйдет из строя). Поддержка конкурентного выполнения в языках программирования имелась, но была очень ограниченной и многими смертными программистами считалась черной магией. Херб Саттер предсказал в своей статье, что «языки программирования… будут вынуждены реализовать все лучшие инструменты поддержки конкуренции».

А теперь посмотрим, какие изменения произошли за десятилетие! Вернувшись в сегодняшний день, мы сразу замечаем приложения, выполняющиеся на большом количестве серверов в облаке, интегрирующем множество систем в разных вычислительных центрах. Все возрастающие потребности конечных пользователей подталкивают требования к производительности и стабильности создаваемых вами систем.

И в чем заключаются эти новые средства поддержки конкуренции? Поддержка конкурентного выполнения в большинстве языков программирования, и особенно в JVM, почти не изменилась. Детали прикладного интерфейса (API) механизма конкуренции значительно усовершенствовались, но программисту все еще приходится работать с низкоуровневыми конструкциями, такими как потоки выполнения и блокировки, которые, как известно, довольно сложны в обращении.

Помимо вертикального масштабирования (увеличения объема вычислительных ресурсов; например количества процессоров на имеющихся серверах), есть также возможность осуществлять *горизонтальное масштабирование* – добавлять в кластер новые серверы. Поскольку с 90-х мало что изменилось в поддержке сетевых взаимодействий языками программирования, многие технологии по-прежнему используют механизм RPC (Remote Procedure Calls – вызовы удаленных процедур) для обмена данными через сеть.

Между тем успехи в организации облачных услуг и создании процессоров с многоядерной архитектурой сделали вычислительные ресурсы еще более доступными.

Предложения PaaS (Platform as a Service – платформа как служба) упростили создание и развертывание очень больших распределенных приложений, которые когда-то могли себе позволить только крупнейшие игроки в индустрии информационных технологий (ИТ). Облачные услуги, такие как AWS EC2 (Amazon Web Services Elastic Compute Cloud) и Google Compute Engine, дают возможность за минуту развернуть буквально тысячи серверов, а поддержка таких инструментов, как Docker, Puppet, Ansible и многих других, упростит управление приложениями на виртуальных серверах.

Количество ядер в процессорах также постоянно растет: даже мобильные телефоны и планшетные компьютеры стали оснащаться многоядерными процессорами.

Но это не означает, что вы можете использовать сколько угодно ресурсов для решения своих задач. В конце концов, все сводится к стоимости и эффективности. То есть приложения должны обеспечивать максимальную эффективность масштабирования, чтобы вложенные деньги окупили себя. Вы никогда не будете использовать алгоритм сортировки с экспоненциальной временной сложностью, и точно так же вы должны задумываться о стоимости масштабирования.

С масштабированием приложений мы обычно связываем два ожидания:

- поскольку добиться соответствия возрастающим требованиям с конечным объемом ресурсов практически нереально, в идеале мы рассчитываем на более медленный рост потребностей в ресурсах, чем требований к приложению. На рис. 1.1 изображена зависимость между требованиями и необходимыми ресурсами;

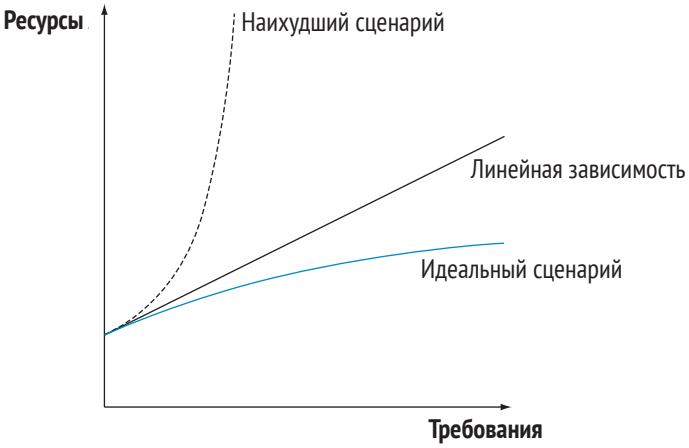

Рис. 1.1. Зависимость требований от ресурсов

- в идеале, если возникает необходимость в наращивании ресурсов, хотелось бы, чтобы сложность приложения оставалась на прежнем уровне или хотя бы росла как можно медленнее. (Вспомните бесплатный сыр в прошлом, когда для ускорения приложения не требовалось увеличивать его сложность!) На рис. 1.2 изображена зависимость между объемом ресурсов и сложностью приложения.

Объем требуемых ресурсов и сложность приложения составляют основную стоимость масштабирования.

Мы исключили множество факторов из этих упрощенных расчетов, но легко видеть, что обе эти составляющие вносят наибольший вклад в общую стоимость масштабирования.

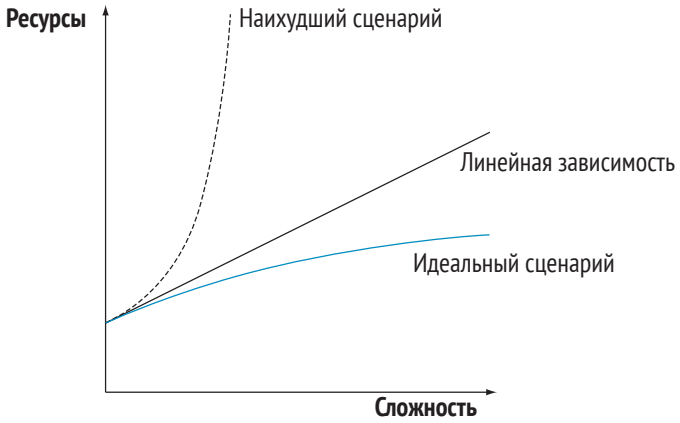

Рис. 1.2. Зависимость сложности от объема требуемых ресурсов

Один их наихудших сценариев – когда приходится платить все больше и больше за все более недоиспользуемые ресурсы. Другой жуткий сценарий – когда сложность приложения переливается через край с добавлением новых ресурсов.

В связи с этим возникает две цели при масштабировании приложения: сложность должна оставаться на как можно более низком уровне и ресурсы должны использоваться максимально эффективно.

Можно ли для достижения этих целей использовать привычные инструменты (потоки и RPC)? Масштабирование по горизонтали с применением RPC и по вертикали с применением низкоуровневых потоков выполнения – не лучший выход. Механизм RPC делает сетевые вызовы не отличимыми от вызовов локальных методов. Каждому вызову RPC приходится блокировать текущий поток выполнения и ждать ответа из сети для работы абстракции вызова локального метода, что может обходиться слишком дорого. Это препятствует эффективному использованию ресурсов.

Другая проблема заключается в необходимости точно знать, как будет выполняться масштабирование – по горизонтали или по вертикали. Многопоточное программирование и сетевое программирование на основе RPC – это как яблоки и груши: они применяются в разных контекстах, используют разную семантику и работают на разных уровнях абстракции. Вам приходится жестко определять, какие части приложения используют потоки для вертикального масштабирования, а какие – RPC для масштабирования по горизонтали.

Наличие жестко заданных методов, действующих на разных уровнях абстракции, значительно увеличивает сложность. А теперь подумайте, что проще – программирование с применением двух запутанных программных конструкций (RPC и потоков) или только с одной конструкцией? Многопрофильный подход к масштабированию намного сложнее, чем необходимо для гибкой адаптации к изменяющимся требованиям.

Запустить тысячу серверов сегодня проще простого, но, как вы увидите в этой первой главе, этого нельзя сказать об их программировании.

1.1. Что такое Akka?

В этой книге мы покажем, как инструменты Akka, открытого проекта, разрабатываемого компанией Lightbend, обеспечивают более простую модель программирования конкурентных и распределенных приложений – *модель акторов*. Акторы не являются чем-то принципиально новым. Они представляют способ масштабирования приложений для JVM в обоих направлениях – по вертикали и по горизонтали. Как вы увидите далее, Akka эффективно использует ресурсы и позволяет сохранить сложность масштабируемых приложений на относительно низком уровне.

Главной целью Akka является простота создания приложений, развертываемых в облаке или запускаемых на многоядерных процессорах, которые эффективно используют все имеющиеся вычислительные ресурсы. Этот фреймворк предоставляет модель акторов, среду времени выполнения и инструменты, необходимые для создания масштабируемых приложений.

1.2. Акторы: краткий обзор

Основу Akka составляют акторы. Большинство компонентов в Akka тем или иным способом поддерживают работу с акторами, будь в том числе средства настройки акторов, соединения акторов с сетью, управления работой акторов и создания кластеров из акторов. Фреймворк Akka выгодно отличается простотой поддержки и наличием инструментов для создания приложений на основе акторов, благодаря чему вы легко сможете сосредоточиться на мышлении и программировании в терминах акторов.

Акторы можно сравнить с очередями сообщений, не имеющими накладных расходов на установку и настройку брокера сообщений. Они подобны программируемым очередям сообщений, сжатым до микроскопических размеров, – вы легко сможете создать тысячи и даже миллионы акторов. Они «ничего не делают», если не посылают сообщения.

Сообщения – это простые структуры данных, которые нельзя изменить после создания, то есть они *неизменяемы*.

Акторы могут получать сообщения по очереди и выполнять в ответ некоторые действия. В отличие от очередей, они могут также посылать сообщения (другим акторам).

Все свои действия акторы выполняют асинхронно. Проще говоря, вы можете отправить сообщение актору и продолжить работу, не дожидаясь ответа. Акторы не похожи на потоки выполнения, но сообщения, посылаемые им, в какой-то момент пересекают некоторый другой поток. Позднее

вы увидите, как настраиваются связи акторов с потоками, а пока просто знайте, что это не жесткие отношения.

Манифест реактивного программирования

Манифест реактивного программирования «The Reactive Manifesto» (http://www.reactivemanifesto.org/) – это инициатива, цель которой – помочь в создании систем, более надежных, более устойчивых, более гибких и лучше соответствующих требованиям современности. Коллектив разработчиков Akka с самого начала был вовлечен в создание манифеста и фреймворк Akka – это результат воплощения идей, выраженных в этом манифесте.

Драйвером большей части манифеста является эффективное использование ресурсов и возможность автоматического масштабирования приложений (также называется *эластичностью*):

- блокирующий ввод/вывод ограничивает возможности параллельного выполнения, поэтому предпочтительнее использовать неблокирующий ввод/вывод;
- синхронные взаимодействия ограничивают возможности параллельного выполнения, поэтому предпочтительнее использовать асинхронные взаимодействия;
- последовательный опрос требует больше ресурсов, поэтому предпочтительнее использовать подходы, основанные на событиях;
- если выход из строя одного узла может нарушить нормальную работу всех остальных узлов, такая организация является напрасным расходованием ресурсов, поэтому нужно изолировать ошибки (увеличить устойчивость), чтобы исключить вероятность потери всей вашей работы;
- системы должны быть эластичными: с уменьшением объема работы система должна потреблять меньше ресурсов; с увеличением объема работы система должна потреблять больше ресурсов, но не больше необходимого минимума.

Сложность составляет основную часть стоимости, поэтому если вы не можете что-то протестировать, изменить или запрограммировать – у вас большие проблемы.

(Перевод «The Reactive Manifesto» на русский язык можно найти по адресу: http://mrdekk.ru/2014/10/17/reactive-manifesto-rus/. – *Прим. перев.*)

Мы детально исследуем акторы в этой книге, но сейчас самое важное, что вы должны понять: принцип действия приложений на основе акторов основан на отправке и приеме сообщений. Сообщения могут обрабаты-

ваться локально, в параллельном потоке выполнения, или удаленно, на другом сервере. Точное место обработки сообщения и где должен находиться актор – все это можно решить позднее. Этим модель акторов выгодно отличается от непосредственного использования потоков выполнения и сетевых взаимодействий в стиле RPC. Акторы упрощают сборку приложений из мелких компонентов, напоминающих сетевые службы, сжатые до микроскопических размеров в смысле занимаемого места и административных издержек.

1.3. Два подхода к масштабированию: подготовка примера

В оставшейся части главы мы исследуем пример коммерческого приложения чата и посмотрим, с какими сложностями приходится сталкиваться при попытке масштабировать его за счет увеличения количества серверов (и при необходимости одновременно обрабатывать миллионы событий). Мы рассмотрим традиционное решение, которое, вероятно, знакомо вам из опыта создания подобных приложений (с применением потоков выполнения и блокировок, RPC и т. п.), и сравним его с подходом на основе использования Akka.

Демонстрацию традиционного подхода мы начнем с простого приложения, размещающегося в памяти, которое затем превращается в приложение, целиком и полностью опирающееся на базу данных как для поддержки конкуренции, так для хранения изменяемого состояния. Чтобы повысить интерактивность такого приложения, нам не остается ничего иного, как организовать опрос базы данных. Мы покажем, что сочетание базы данных и RPC-взаимодействий влечет плохо контролируемое увеличение сложности приложения. Мы также покажем, что изолировать ошибки в этом приложении становится все труднее и труднее с каждым новым шагом вперед. Мы уверены, что многие из этих проблем давно знакомы вам.

Затем мы посмотрим, как модель акторов упрощает приложение и как фреймворк Akka помогает написать приложение однажды и масштабировать его при любой возможности (автоматически решая проблемы конкурентного выполнения). В табл. 1.1 перечислены различия между двумя подходами. Некоторые пункты пока будут непонятны, но мы проясним их в следующих разделах.

Вообразите, что мы решили покорить мир с помощью современного приложения чата, революционизирующего область онлайн-сотрудничества. Его цель – помогать членам одной команды быстро находить друг друга и работать вместе. У нас очень много идей, которые мы могли бы реализовать в этом приложении, в том числе интеграция с инструментами управления проектами и с существующими службами связи.

Таблица 1.1. Различия между подходами

Цели	Методы достижения	
	Традиционный	**На основе Akka**
Масштабирование	Использовать комбинацию потоков выполнения, общего изменяемого состояния в базе данных (операции создания, добавления, изменения, удаления) и RPC-вызовов веб-служб	Акторы посылают и принимают сообщения. Общее изменяемое состояние отсутствует. Выполнением управляют неизменяемые события
Передача интерактивной информации	Опрос текущей информации	Передача при появлении события
Масштабирование в сети	Синхронные RPC-вызовы, блокирующий ввод/вывод	Асинхронная передача сообщений, неблокирующий ввод/вывод
Обработка ошибок	Обработка всех исключений; работа продолжается, только если все компоненты сохраняют работоспособное состояние	Ошибка в одном компоненте не имеет катастрофических последствий. Отказы изолируются, и работа продолжается без отказавших компонентов

В духе Lean Startup[1] начнем с создания продукта с минимальной ценностью, чтобы лучше узнать наших потенциальных пользователей и понять их потребности. Если он взлетит, мы сможем заполучить миллионы пользователей. И мы знаем, что есть две силы, способные замедлить наше движение вперед вплоть до остановки.

- *Сложность* – приложение становится все сложнее, и в него все труднее добавлять новые возможности. Даже небольшие изменения требуют приложения значительных усилий, и их все труднее и труднее тестировать.
- *Жесткость* – приложение плохо приспосабливается; с каждым большим шагом в приросте числа пользователей его приходится переписывать с нуля. Переписывание требует времени и само является сложным процессом. Как только число пользователей становится больше, чем мы можем обслужить, нам приходится разрываться между поддержкой работоспособности существующей версии приложения и созданием новой, способной обслуживать больше пользователей.

У нас уже есть некоторый опыт создания приложений, и мы решили пойти уже изученным путем, избрав традиционный подход с использованием низкоуровневых потоков выполнения с блокировками, RPC, блокирующим вводом/выводом и изменяемым состоянием в базе данных.

[1] https://ru.wikipedia.org/wiki/Бережливый_стартап. – *Прим. перев.*

1.4. Традиционное масштабирование

Начнем с создания сервера. Для первой версии приложения чата мы придумали модель данных, изображенную на рис. 1.3. Пока мы просто будем хранить эти объекты в памяти.

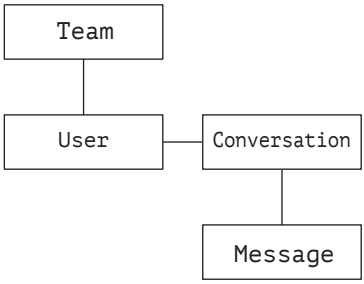

Рис. 1.3. Модель данных

`Team` (команда) – это группа объектов `User` (пользователь). Несколько объектов `User` могут одновременно участвовать в некотором диалоге `Conversation`. Диалоги `Conversation` – это коллекции сообщений `Message`. Пока все хорошо.

На основе этой модели мы реализовали поведение приложения и сконструировали пользовательский веб-интерфейс. Теперь мы можем показать приложение потенциальным пользователям и провести демонстрацию его возможностей. Код приложения прост и легко управляем. Но пока приложение работает исключительно в памяти, поэтому всякий раз, когда оно перезапускается, все диалоги теряются. Кроме того, на данном этапе приложение может выполняться только на одном сервере. Пользовательский веб-интерфейс нашего приложения, созданный с помощью [вставьте сюда название лучшей, по вашему мнению, библиотеки JavaScript], оказался настолько впечатляющим, что заинтересованные лица пожелали немедленно начать использовать его, хотя мы неоднократно предупреждали их, что это всего лишь демонстрационный вариант! Пришло время увеличить количество серверов и настроить эксплуатационное окружение.

1.4.1. Традиционный подход к масштабированию и хранению: переместить все в базу данных

Мы решили добавить в уравнение базу данных. Мы планируем запустить приложение на двух веб-серверах для большей доступности с балансировщиком нагрузки перед ними. На рис. 1.4 показано, как выглядит эта конфигурация.

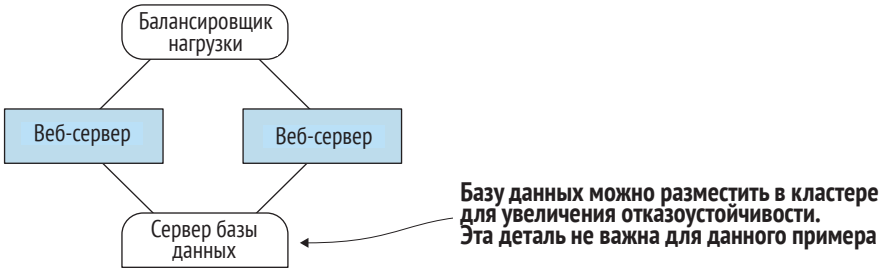

Рис. 1.4. Балансировщик нагрузки

Код стал сложнее, потому что теперь мы не можем продолжать держать объекты в памяти, иначе как бы мы обеспечили их непротиворечивость на двух серверах? Кое-кто из нашей команды воскликнул: «Мы должны избавиться от хранения состояния в памяти!» – и мы заменили все основные объекты кодом для работы с базой данных.

Состояние объектов больше не хранится в памяти веб-серверов, а это означает, что методы объектов больше не могут обращаться к состоянию непосредственно; вся важная логика переместилась в инструкции в базе данных. Изменения изображены на рис. 1.5.

Рис. 1.5. Объекты доступа к данным

Отказ от хранения состояния в памяти привел к решению заменить объекты некоторой абстракцией доступа к базе данных. Выбор абстракции не имеет большого значения для нашего примера; в данном случае мы решили не отступать от традиций и использовали объекты доступа к данным (Data Access Objects, DAO), которые выполняют инструкции в базе данных.

Из-за этого многое что изменилось.

- Мы утратили некоторые гарантии, которые имели прежде, например при вызове метода объекта `Conversation`, добавляющего новое сообщение `Message`. Прежде мы могли гарантировать, что вызов `addMessage` никогда не потерпит неудачу, потому что выполнял простейшую операцию со списком в памяти (исключая ситуацию исчерпания памяти в JVM). Теперь же база данных может вернуть признак ошибки

в ответ на любой вызов `addMessage`. Инструкция вставки может потерпеть неудачу, база данных может оказаться недоступной в этот момент из-за проблем с сетью или из-за выхода из строя сервера.

- В версии, где все данные хранились в памяти, использовалось множество блокировок, препятствующих повреждению данных одновременно работающими пользователями. Теперь, перейдя на использование базы данных, мы должны выяснить, как справиться с этой проблемой и гарантировать невозможность создания дубликатов записей или появления других противоречивых данных. Каждый вызов метода теперь фактически превращается в операции, происходящие в базе данных, которые должны выполняться в определенной последовательности. Например, чтобы начать диалог, требуется добавить записи в две таблицы – `Conversation` и `Message`.
- Версия в памяти легко поддавалась тестированию, и тесты выполнялись очень быстро. Теперь для тестирования используется локальная база данных, и мы добавили несколько утилит для изоляции тестов. Тестирование уже выполняется намного медленнее. Но мы утешаем себя: «Зато мы тестируем также и операции с базой данных».

Преобразование кода, прежде работавшего с памятью, в обращения к базе данных может стать причиной низкой производительности, потому что каждый вызов теперь приводит к обмену данными через сеть. Поэтому мы спроектировали структуры данных так, чтобы оптимизировать производительность запросов для выбранной базы данных (SQL или NoSQL, что совершенно не важно). Теперь объекты являются бледными подобиями их прежних «я» и просто хранят данные; весь основной код переместился в объекты DAO и компоненты веб-приложения. Самое неприятное во всем этом – мы теперь едва ли сможем повторно использовать прежний код; структура кода полностью изменилась.

«Контроллеры» в веб-приложении комбинируют методы объектов DAO для изменения данных (`findConversation`, `insertMessage` и т. д.). Такое комбинирование методов порождает взаимодействия с базой данных, которые трудно предсказать; контроллеры могут свободно создавать любые комбинации операций с базой данных, как показано на рис. 1.6.

На рис. 1.6 представлен один из возможных сценариев – добавление сообщения `Message` в диалог `Conversation`. Вообразите, что существует большое количество сценариев доступа к базе данных посредством объектов DAO. Если позволить любой части приложения свободно изменять или запрашивать записи в любой момент времени, это может привести к трудно предсказуемым проблемам производительности, таким как взаимоблокировки и др. Именно этой сложности мы хотели бы избежать.

Обращение к базе данных, по сути, является вызовом RPC, и почти все стандартные драйверы баз данных (такие как JDBC) используют блокиру-

ющий ввод/вывод. То есть мы оказались в ситуации, описанной выше: мы одновременно используем потоки выполнения и вызовы RPC. Блокировки памяти, используемые для синхронизации потоков, и блокировки в базе данных, защищающие записи от недопустимых изменений, – это далеко не то же самое, и мы должны проявить максимум осторожности, объединяя их. Мы перешли от одной к двум, тесно переплетенным моделям программирования.

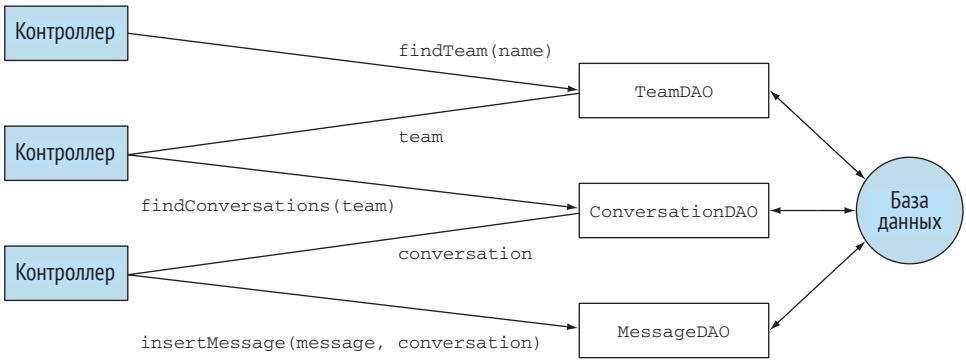

Рис. 1.6. Взаимодействие с базой данных через объекты DAO

Мы только что произвели первую переделку приложения, и для этого нам потребовалось больше времени, чем ожидалось.

> **ЭТО БОЛЬШАЯ ПРОБЛЕМА.** Традиционный подход к реализации приложения коллективного чата оказался никуда негодным. Да, мы несколько сгустили краски, но вы наверняка видели проекты, сталкивающиеся хотя бы с некоторыми из этих проблем (уж мы так точно наблюдали все это в своей практике). Как сказал Дин Уэмплер (Dean Wampler) в своей презентации «Reactive Design, Languages and Paradigms» (https://deanwampler.github.io/polyglotprogramming/papers/):
>
> *В действительности люди могут заставить работать любое решение, даже не самое оптимальное.*
>
> То есть данный пример проекта невозможно завершить, используя традиционный подход? Нет, но результат получится далеко не оптимальным. Будет очень сложно поддерживать низкую сложность и высокую гибкость при масштабировании приложения.

1.4.2. Традиционное масштабирование и интерактивная работа: опрос

Мы внедрили эту конфигурацию, и круг пользователей приложения начал расширяться. Веб-серверы приложения используют не так много ре-

сурсов; большая их часть тратится на (де)сериализацию запросов и ответов. Основное время расходуется на взаимодействия с базой данных. Код, выполняющийся на веб-сервере, большую часть времени проводит в ожидании ответа от драйвера базы данных.

Но теперь, когда у нас появился фундамент, хотелось бы добавить больше интерактивных функций. Пользователи привыкли к Facebook и Twitter и хотели бы получать уведомления, когда их имена упоминаются в диалоге, чтобы присоединиться к беседе.

Нам нужно реализовать компонент `Mentions`, анализирующий все сообщения и добавляющий упоминаемые контакты в таблицу уведомлений, которая затем будет опрашиваться веб-приложением и извещать упомянутых пользователей.

Теперь веб-приложение должно так же чаще опрашивать другую информацию, чтобы быстрее отразить изменения в сведениях о пользователях, потому что наша цель – дать им по-настоящему интерактивный опыт.

Мы решили не замедлять работу диалогов и не добавлять код для работы с базой данных непосредственно в приложение, а организовать очередь сообщений. Каждое сообщение асинхронно записывается в эту очередь, а отдельный процесс извлекает их из очереди, отыскивает упоминания пользователей и при необходимости создает записи в таблице уведомлений.

На данный момент база данных действительно оказалась перегружена. Мы заметили, что автоматизированный опрос базы данных компонентом `Mentions` вызывает проблемы с производительностью базы данных. Поэтому мы выделили компонент `Mentions` в отдельную службу и дали ему свою базу данных, содержащую таблицу с уведомлениями и копию таблицы, содержащей информацию о пользователях, обновляемую заданием в базе данных, как показано на рис. 1.7.

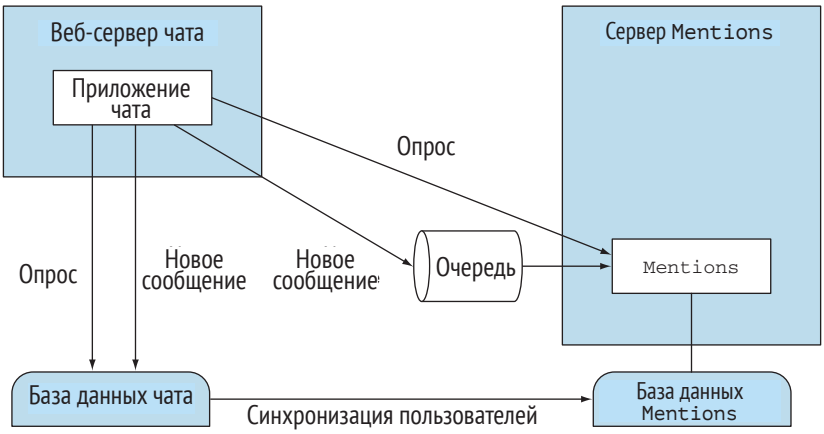

Рис. 1.7. Компонент службы

Мало того, что сложность снова увеличилась, стало сложнее добавлять новые интерактивные функции. Опрос базы данных, как оказалось, далеко не самая лучшая идея для такого рода приложений, но в подобном случае нет иных альтернатив, потому что вся логика сосредоточена в объектах доступа к данным и сама база данных не может послать событие веб-серверу.

Кроме того, внедрение очереди сообщений также усложнило приложение. Эту очередь требуется установить и настроить, и необходимо развернуть код, обслуживающий ее. Очередь сообщений имеет свою семантику и контекст; она действует иначе, чем RPC-вызовы базы данных или вызовы многопоточного кода в памяти. Сочетание всех этих сложностей еще больше увеличивает сложность приложения.

1.4.3. Традиционное масштабирование и интерактивная работа: обработка ошибок

Пользователи начинают сообщать в отзывах, что им хотелось бы иметь возможность находить контакты *во время ввода* (когда приложение дает подсказки, пока пользователь продолжает ввод имени контакта) и автоматически получать подсказки для групп и текущих диалогов, опираясь на недавнее общение по электронной почте. С этой целью мы создали объект `TeamFinder`, который обращается к нескольким веб-службам, таким как Google Contacts API и Microsoft Outlook.com API. Мы создали для этого отдельную веб-службу клиентов и внедрили поиск контактов, как показано на рис. 1.8.

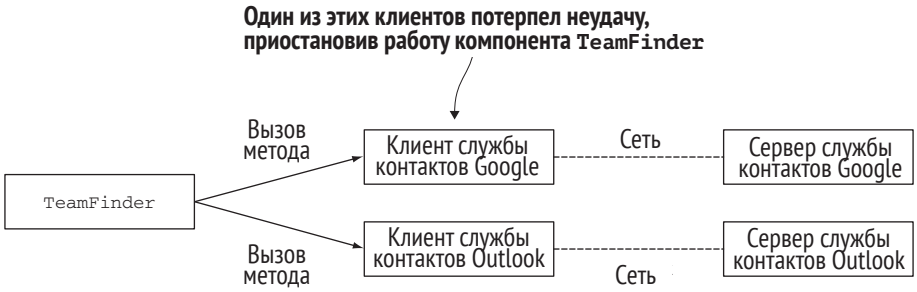

Рис. 1.8. Поиск контактов

В процессе работы мы выяснили, что одна из служб часто терпит неудачу и в результате наш компонент простаивает долгое время, ожидая окончания тайм-аута. Иногда трафик оказывается настолько большим, что обмен с внешней службой замедляется до нескольких байтов в минуту. И поскольку обращения к службам следуют один за другим, поиск слишком долго ждет ответа, даже притом, что мог бы получить множество полезных подсказок от службы, которая работает нормально.

Хуже того, несмотря на то что методы обращения к базе данных мы сосредоточили в объектах DAO, а поиск контактов в объекте `TeamFinder`, контроллеры вызывают эти методы как любые другие. Это означает, что иногда поиск происходит между двумя обращениями к базе данных, из-за чего соединение остается открытым дольше, чем хотелось бы, потребляя ресурсы базы данных. Если `TeamFinder` терпит неудачу, все остальные этапы в том же потоке приложения также терпят неудачу. Контроллер возбуждает исключение и оказывается не в состоянии продолжить работу. Можно ли каким-то образом надежно отделить `TeamFinder` от остального кода?

Пришло время снова переписать приложение, и не похоже, что сложность на этот раз уменьшится. Фактически теперь мы используем четыре модели программирования: одна для управления потоками выполнения в памяти, одна для выполнения операций с базой данных, одна для очереди сообщений `Mentions` и одна для работы с веб-службами контактов.

Как в такой ситуации задействовать, скажем, 10 серверов или 100 вместо 3, если потребуется? Очевидно, что выбранный нами подход плохо масштабируется: мы вынуждены менять направление с каждой новой проблемой.

В следующем разделе вы познакомитесь со стратегией, которая не требует изменения направления с каждой новой проблемой.

1.5. Масштабирование с Akka

Давайте посмотрим, можно ли только с помощью акторов удовлетворить требования к масштабированию приложения. Поскольку пока еще не совсем ясно, как именно работают акторы, будем использовать объекты и акторы взаимозаменяемо, и сосредоточимся на концептуальной разнице между этим и традиционным подходами.

Различия между подходами перечислены в табл. 1.2.

Таблица 1.2. Сравнение акторов и традиционного подхода

Цели	Методы достижения	
	Традиционный	**На основе Akka (акторы)**
Обеспечить сохранность диалогов даже в случае перезапуска приложения или его аварийного завершения	Переписать код в объектах DAO. Использовать базу данных как одно большое изменяемое состояние, когда все части приложения могут создавать, изменять, добавлять и удалять данные	Продолжать использовать состояние в памяти. Изменения в состоянии посылаются в виде сообщений в журнал. Журнал приходится перечитывать только в случае перезапуска приложения

Цели	Методы достижения	
	Традиционный	На основе Akka (акторы)
Добавить интерактивные возможности (Mentions)	Опрашивать базу данных. Опрос потребляет значительные ресурсы, даже в отсутствие изменений в данных	Рассылать события в нужные части приложения. Объекты уведомляют друг друга только при появлении событий, что снижает накладные расходы
Отделить службы; служба Mentions и функции чата не должны мешать друг другу	Добавить очередь сообщений для асинхронной обработки	Нет необходимости добавлять очередь сообщений; акторы по определению действуют асинхронно. Никакой дополнительной сложности; вы уже знакомы с принципами отправки/приема сообщений
Предотвратить сбой всей системы, когда происходит отказ важной службы или служба действует слишком медленно	Попытаться предотвратить любые ошибки, предсказывая все возможные ситуации и перехватывая соответствующие исключения	Сообщения передаются асинхронно; если сообщение не было обработано отказавшим компонентом, это не повлияет на стабильную работу других компонентов

Было бы здорово, если бы мы могли написать прикладной код один раз и затем масштабировать его как угодно. Нам важно избежать радикального изменения главных объектов приложения; как, например, произошло, когда мы заменили всю логику работы в памяти объектами DAO в разделе 1.4.1.

Первая проблема, которую мы должны были решить, – сохранение диалогов. Непосредственное использование базы данных отодвинуло нас от простой модели работы с памятью. Превратив методы в RPC-команды базы данных, мы получили смешанную модель программирования. Мы должны найти другой способ гарантировать сохранность диалогов, сохранив приложение максимально простым.

1.5.1. Подход к масштабированию и хранению с Akka: отправка и прием сообщений

Для начала решим самую первую задачу и просто обеспечим сохранность диалогов. Приложение должно сохранять диалоги некоторым способом и восстанавливать их после перезапуска.

На рис. 1.9 показано, как объект `Conversation` посылает событие `MessageAdded` в журнал в ответ на добавление каждого нового сообщения в памяти.

Объект `Conversation` можно восстановить из этих объектов-событий, хранящихся в базе данных, как показано на рис. 1.10.

Точный порядок работы мы обсудим позже. Но уже сейчас вы можете видеть, что база данных используется только для восстановления сообщений в диалогах. Мы не используем ее для выражения операций. Актор

Conversation посылает сообщения в журнал и получает их обратно в момент запуска. Нам не нужно изучать ничего нового; это всего лишь отправка и прием сообщений.

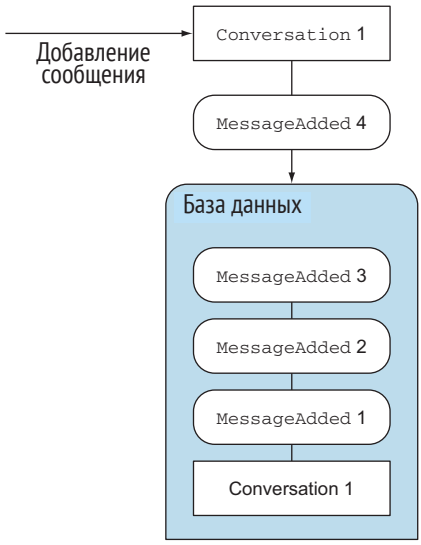

Рис. 1.9. Сохранение диалогов

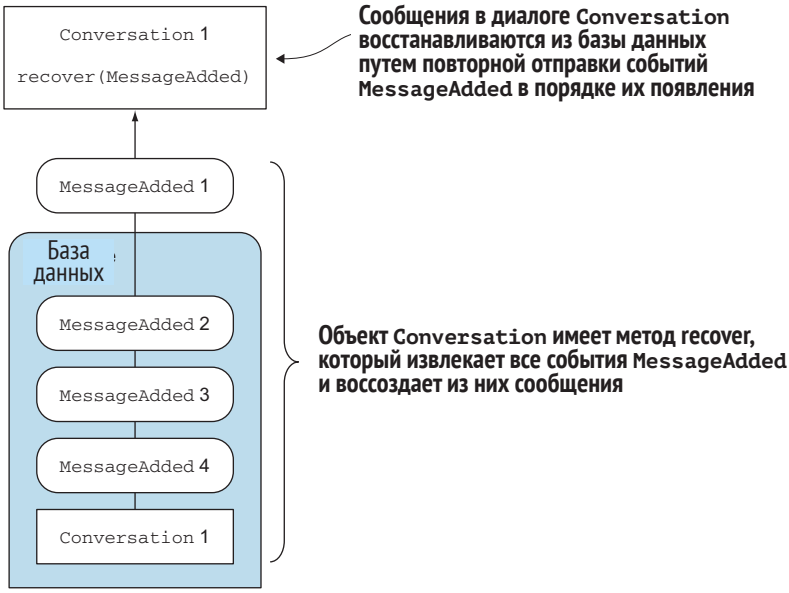

Рис. 1.10. Восстановление диалогов

Изменения сохраняются как последовательность событий

Все изменения сохраняются как последовательность событий, в данном случае событий MessageAdded. Текущее состояние диалога Conversation

можно восстановить, воспроизведя события, возникавшие в памяти `Conversation`, поэтому мы легко можем организовать возобновление работы с места остановки. Базы данных такого вида часто называют *журналами*, а сам прием – *порождение событий*. О приеме порождения событий можно рассказывать долго, но пока ограничимся этим определением.

Важно также отметить, что журнал в данной реализации превратился в универсальную службу. Все, что нужно, – сохранять все события по мере их появления и затем, когда понадобится, извлечь их в том же порядке, в каком они были записаны в журнал. Есть еще некоторые тонкости, которые мы пока не будем затрагивать, такие как сериализация, но если вам не терпится, загляните в главу 15, где рассказывается о хранимых акторах.

Распределение данных: фрагментирование диалогов

Следующая наша проблема – мы все еще кладем все яйца в одну корзину, то бишь сервер. Когда сервер перезапускается, он читает все диалоги в память и продолжает работу. Главная причина, почему мы отказались от хранения состояния в памяти при реализации традиционного подхода, состояла в том, что нам трудно было представить, как обеспечить синхронизацию диалогов между несколькими серверами. Но что случится, если на одном сервере окажется слишком много диалогов?

Решить эту проблему можно, распределив диалоги по серверам предсказуемым образом или запоминая, где находится каждый диалог. Этот прием называется *шардингом* (sharding), или *фрагментированием*. На рис. 1.11 показано несколько диалогов, распределенных между двумя серверами.

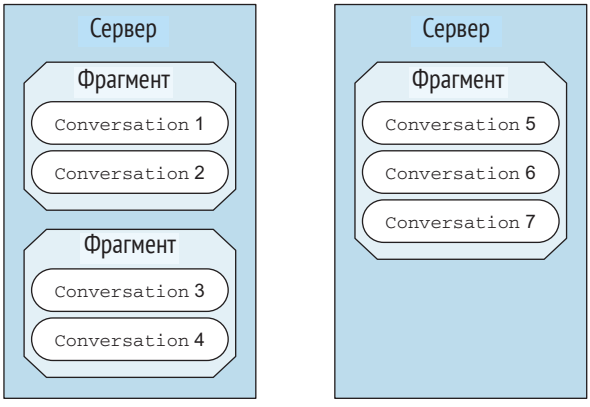

Рис. 1.11. Фрагментирование

Мы сможем продолжать использовать простую модель диалогов в памяти, если у нас будет универсальный журнал для хранения событий и способ определить фрагментирование диалогов. Более подробно о деталях этих

двух механизмов мы поговорим в главе 15. А пока представим, что мы можем просто использовать эти службы.

1.5.2. Масштабирование с Akka и интерактивная работа: отправка сообщений

Вместо опроса базы данных для каждого пользователя веб-приложения мы могли бы найти способ уведомить пользователя о важных изменениях (событиях), напрямую посылая сообщения веб-браузеру пользователя.

Приложение может также посылать сообщения внутри себя, как признак выполнения определенных задач. Каждый объект в приложении будет посылать событие, когда в нем будет происходить что-то интересное. Другие объекты смогут принимать события и выполнять ответные действия, как показано на рис. 1.12.

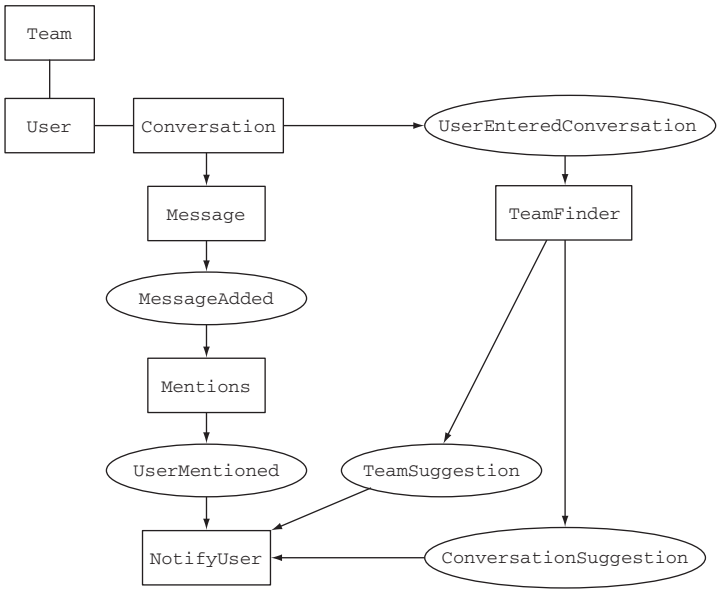

Рис. 1.12. События

События (на рис. 1.12 изображены как эллипсы) устраняют из системы нежелательные тесные связи между компонентами. Диалог `Conversation` просто публикует событие в ответ на добавление сообщения `Message` и продолжает свою работу. События распространяются с использованием механизма публикации/подписки, без прямых взаимодействий компонентов друг с другом. Событие в конечном итоге попадает подписчикам, в данном случае – компоненту `Mentions`. Еще раз отметим, что решение этой проблемы можно смоделировать, просто посылая и принимая сообщения.